НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК _____

«До захисту допущено»

Завідувач кафедри СПСКС

_____ В.П.Тарасенко_
(підпис) (ініціали, прізвище)

"___"_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
(«Системне програмування»)

на тему: «МЕТОД ДИНАМІЧНОЇ КОМПІЛЯЦІЇ PYTHON ПРОГРАМ, ЩО ОРІЄНТОВАНІ НА ОБРОБКУ МАСИВІВ»

Виконав: студент II курсу, групи КВ-62м
(шифр групи)

Кривомаз Максим Євгенович _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник д.т.н., доцент Терейковський І.А _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
Системне програмування

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ _В.П.Тарасенко__
(підпис) (ініціали, прізвище)

«___»_____2018р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Кривомаза Максима Євгеновича
(прізвище, ім'я, по батькові)

1. Тема дисертації «Метод динамічної компіляції Python програм, що орієнтовані на обробку масивів»,
науковий керівник дисертації д.т.н., доцент Терейковський І. А. ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «22» березня 2018 р. №986-с

2. Термін подання студентом дисертації 11 травня 2018 р.

3. Об'єктом дослідження є метод динамічної компіляції Python програм, що орієнтовані на обробку масивів.

4. Предметом дослідження є методи і алгоритми виконання програмного коду та його оптимізації.

5. Перелік завдань, які потрібно зробити

- провести порівняльний аналіз існуючих методів виконання та інтерпретації програмного коду.

- дослідити алгоритми і методи оптимізації програмного коду, а також технології, що можуть бути застосовані для виконання програмного коду у паралельних потоках.

- описати та розглянути переваги використання динамічної компіляції у інтерпретуємих мовах програмування.

- запровадити паралелізм обробки даних для операторів роботи над масивами.

- запропонувати та розглянути метод динамічної компіляції у поєднанні з рядом оптимізацій різного рівня та паралелізмом обробки даних.

- розглянути етапи перетворення програмного коду функції, що оптимізується, у проміжних представленнях.

- провести експерименти на ряді алгоритмів з інтенсивними обчисленнями, порівняти швидкість їх виконання стандартними засобами мови Python та з використанням запропонованого методу, провести аналіз отриманих результатів у залежності від вибраного бекенду виконання програмного коду.

6. Перелік ілюстративного матеріалу

- Структурна схема процесу інтерпретації програмного коду з використанням запропонованого методу

- Схема алгоритму оптимізації програмного коду функції перед утворенням проміжного нетипізованого представлення

- Схема алгоритму оптимізації нетипізованого представлення після виклику функції

- Схема алгоритму роботи запропонованого методу

- Схема алгоритму типізації даних у функції

- Демонстраційні гістограми результатів тестування методу на наборі алгоритмів

7. Перелік публікацій

- Тези доповіді «Динамічна компіляція Python-програм в задачах захисту інформації»

- Тези доповіді «Спосіб динамічної компіляції масивно орієнтованих Python програм»

8. Дата видачі завдання 5 вересня 2016 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Ґрунтовне ознайомлення з предметною областю дослідження	17.12.2016	
2	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури	04.03.2017	
3	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	16.05.2017	
4	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	14.10.2017	
5	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	15.12.2017	
6	Проведення наукового дослідження; робота над третім розділом магістерської дисертації; підготовка матеріалів доповіді на конференції ПМК-2018.	20.02.2018	
7	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу;	16.04.2018	
8	Оформлення текстової і графічної частини магістерської дисертації	20.04.2018	
9	Попередній розгляд магістерської дисертації на кафедрі	26.04.2018	

Студент _____ _____
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ _____
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Актуальність теми. Мова програмування Python стала популярною платформою для аналізу даних і наукових обчислень. Python транслює інструкції вихідного програмного коду в проміжне представлення, відоме як байт-код, і потім інтерпретує цей байт-код. Байт-код забезпечує переносимість програм, оскільки це платформо-незалежний формат. Однак через те, що Python не створює двійковий машинний код (наприклад, машинні інструкції для мікропроцесора Intel), деякі програми на мові Python можуть працювати повільніше своїх аналогів, написаних на мовах що компілюються у машинний код, таких як C. Для вирішення проблеми низької продуктивності стандартного інтерпретатора мови Python, математично інтенсивні обчислення, як правило, переносяться на бібліотечні функції, які написані на високопродуктивних мовах програмування. Якщо така бібліотека для виконання певного алгоритму відсутня, програмісту доводиться прийняти низьку продуктивність або перейти на мову нижчого рівня для ефективної реалізації поставленої задачі.

Таким чином, у процес розробки проекту входить етап прототипування алгоритмів на мови програмування нижчого рівня, а потім безпосередньо відбувається процес переносу розділів з низькою продуктивністю у такій мові як Python, на мову нижчого рівня. Цей етап може займати багато часу, призводити до появи помилок і відводить увагу розробника від початкової задачі, тому в даній роботі запропоновано метод динамічної компіляції Python програм орієнтованих на обробку масивів, який завдяки оптимизаціям високого рівня, визначенню типів вхідних даних та використанню переваг паралельного виконання коду надає значне прискорення часу виконання програм, які виконують операції над масивами.

Об’єктом дослідження є метод динамічної компіляції Python програм, що орієнтовані на обробку масивів.

Предметом дослідження є методи і алгоритми принципів трансляції та виконання програмного коду, оптимізації проміжного представлення програмного коду та оптимізації компіляції програм, що орієнтовані на обробку масивів.

Мета і задачі дослідження: створити метод динамічної компіляції Python програм, що орієнтовані на обробку масивів, для вирішення проблеми низької продуктивності математично інтенсивних обчислень над масивами у мові Python та уникнення прототипування алгоритмів на мови нижчого рівня та переносу розділів з низькою продуктивністю на такі мови. Запропонований метод надасть можливість поєднати зручність використання мови Python та забезпечить швидкість виконання коду як на ефективних мовах програмування. Провести експерименти порівняння часу виконання ряду алгоритмів на різних апаратних засобах.

Наукова новизна полягає в наступному:

1. Запропонований метод динамічної компіляції Python програм, що орієнтовані на обробку масивів, значно підвищує швидкодію виконання коду у порівнянні зі стандартним інтерпретатором мови Python.

2. Розроблено програмну реалізацію методу, що поєднує у собі динамічну компіляцію у сукупності з набором оптимізацій різного рівня та реалізацією паралелізму даних та застосовується для ефективної реалізації примітивів та операторів бібліотеки розширення масивів NumPy, яка є незамінним інструментом при виконанні аналізу даних та наукових обчислень мовою Python.

Практична цінність полягає у підвищенні як швидкості виконання програмного коду, так і продуктивності його написання. Метод забезпечує

поєднання зручності використання мови Python та швидкості виконання коду на ефективних мовах програмування.

Апробація роботи.Основні положення і результати роботи будуть представлені та обговорюватимуться на науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018 (Київ, 22-24 березня 2018 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботи, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі розглянуто основні засоби трансляції та виконання програмного коду, зроблено порівняльний аналіз компіляції у машинний код та інтерпретації байт-коду, описано проблематику реалізації алгоритмів з інтенсивними обчисленнями над масивами у мові Python та зроблено короткий огляд самої мови Python.

У другому розділі виконано детальний огляд інтерпретації програмного коду у мові Python, описані методи оптимізації програмного коду та компіляції, що використані у запропонованому методі, розглянуто використання проміжного представлення програмного коду та описано технології за допомогою яких реалізується паралелізм даних.

У третьому розділі описується реалізація методу динамічної компіляції Python програм, що орієнтовані на обробку масивів, детально розглянуто усі етапи перетворення програмного коду функції від початкового нетипізованого проміжного представлення до безпосереднього виконання машинного коду, описані реалізовані оператори паралелізму даних та роботи з масивами.

У четвертому розділі представлено результати проведеного тестування методу на наборі алгоритмів з інтенсивними обчисленнями, що

виконують обробку масивів, зроблено аналіз покращення швидкодії у залежності від вибраного бекенду виконання машинного коду та проведено аналіз факторів від яких залежить оптимізація виконання програмного коду.

У висновках представлені результати проведеної роботи.

Робота представлена на 82 аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: динамічна компіляція, паралелізм даних, масивно-орієнтоване програмування, Python.

ABSTRACT

Relevance of the topic. The Python programming language has become a popular platform for data analysis and scientific computing. Python translates the instructions of the program code into an intermediate representation, known as a bytecode, and then interprets this bytecode. Bytecode ensures portability of programs, because it is platform-independent format. However, because Python does not create binary machine code (for example, machine instructions for the Intel processor), some Python programs can run slower than their counterparts written in languages that are compiled into machine code, such as C. To solve the problem of low performance of the standard Python interpreter, mathematically intensive computations are usually carried over to library functions that are written in high-performance programming languages. If such a library does not exist for executing a certain algorithm, the programmer must accept low performance or switch to a low-level language for effective implementation of the task.

Thus, the process of project development includes the stage of prototyping algorithms into low-level programming languages, and then the process of transferring partitions with low performance in a language such as Python into a low-level language takes place directly. This stage can take a lot of time, lead to errors and draws the attention of the developer from the initial task, so in this paper we propose a method of dynamical compiling of Python array-oriented programs, which, thanks to high-level optimization, determining the types of input data and taking advantage of parallel code execution significantly speeds up the execution time of programs that perform operations on arrays.

The object of research is the method of dynamic compilation of Python programs, oriented to the processing of arrays.

The subject of the study are methods and algorithms for the principles of translating and executing program code, optimizing the intermediate representation of code, and optimizing the compilation of programs oriented to array processing.

The purpose and objectives of the study: to create a method for dynamic compilation of Python array oriented programs to solve the problem of low performance of mathematically intensive computations over arrays in Python and avoid prototyping algorithms to low-level languages and transfer sections with low performance to such languages. The proposed method will combine the convenience of using the Python language and ensure the speed of code execution in both effective programming languages. Perform experiments comparing the execution time of a number of algorithms on various hardware.

Scientific novelty consists in the following:

1. The proposed method of dynamic compilation of Python programs, oriented to the processing of arrays, significantly improves the speed of code execution compared to the standard interpreter of Python.
2. A software implementation of a method combining dynamic compilation in combination with a set of optimizations of different levels and the implementation of data parallelism is developed and applied to the effective implementation of primitives and operators of the NumPy array extension library, which is an indispensable tool for performing data analysis and scientific calculations in Python.

The practical value is to increase both the speed of execution of the program code, and the productivity of its writing. The method provides a combination of the convenience of using the Python language and the speed of code execution in effective programming languages.

Approbation of method. The main provisions and results of the work will be presented and will be discussed at the scientific conference of undergraduates and postgraduates "Applied Mathematics and Computing" PMK-2018 (Kiev, March 22-24, 2018).

Structure and amount of work. The master's thesis consists of an introduction, four chapters and conclusions.

The introduction presents a general description of the work, the relevance of the research direction is substantiated, the goals and objectives of the research

are formulated, the scientific novelty of the results obtained and the practical value of the work are shown.

In the first section, the main tools for translating and executing program code are considered, a comparative analysis of compilation into machine code and interpretation of bytecode is made, the problems of implementing algorithms with intensive computations over arrays in Python are described and a brief overview of the Python language itself is made.

The second section provides a detailed overview of the interpretation of code in Python, describes the methods for optimizing code and compilation used in the proposed method, examines the use of an intermediate representation of the code, and describes the technologies by which data parallelism is realized.

The third section describes the implementation of the method of dynamic compilation of Python programs, oriented to the processing of arrays, details all the stages of converting the program code of the function from the initial untyped intermediate representation to the immediate implementation of the machine code, the described operators implemented data parallelism and work with arrays.

The fourth section presents the results of testing the method on a set of algorithms with intensive calculations that perform array processing, analyzing the performance improvement depending on the selected backend for executing the machine code, and analyzing the factors on which optimization of code execution depends.

The conclusions contain the results of the work.

The work is presented on 82 pages, contains references to a list of literary sources used.

Keywords: dynamic compilation, data parallelism, massively-oriented programming, Python.

РЕФЕРАТ

Актуальность темы. Язык программирования Python стал популярной платформой для анализа данных и научных вычислений. Python транслирует инструкции програмного кода в промежуточное представление, известное как байт-код, и затем интерпретирует этот байт-код. Байт-код обеспечивает переносимость программ, поскольку это платформо-независимый формат. Однако из-за того, что Python не создает двоичный машинный код (например, машинные инструкции для процессора Intel), некоторые программы на языке Python могут работать медленнее своих аналогов, написанных на языках которые компилируются в машинный код, таких как C. Для решения проблемы низкой производительности стандартного интерпретатора языка Python, математически интенсивные вычисления, как правило, переносятся на библиотечные функции, которые написаны на высокопроизводительных языках программирования. Если такая библиотека для выполнения определенного алгоритма отсутствует, программисту приходится принять низкую производительность или перейти на язык низкого уровня для эффективной реализации поставленной задачи.

Таким образом, в процесс разработки проекта входит этап прототипирования алгоритмов на языки программирования низкого уровня, а затем непосредственно происходит процесс переноса разделов с низкой производительностью в таком языке как Python, на язык низкого уровня. Этот этап может занимать много времени, приводить к появлению ошибок и отводит внимание разработчика от начальной задачи, поэтому в данной работе предложен метод динамической компиляции Python программ ориентированных на обработку массивов, который благодаря оптимизации высокого уровня, определению типов входных данных и использованию преимуществ параллельного выполнения кода оказывает существенное ускорение времени выполнения программ, которые выполняют операции над массивами.

Объектом исследования является метод динамической компиляции Python программ, ориентированных на обработку массивов.

Предметом исследования являются методы и алгоритмы принципов трансляции и выполнения программного кода, оптимизации промежуточного представления кода и оптимизации компиляции программ, ориентированных на обработку массивов.

Цель и задачи исследования: создать метод динамической компиляции Python программ, ориентированных на обработку массивов, для решения проблемы низкой производительности математически интенсивных вычислений над массивами в языке Python и избежания прототипирования алгоритмов на языки низкого уровня и переноса разделов с низкой производительностью на такие языки. Предложенный метод позволит совместить удобство использования языка Python и обеспечит скорость выполнения кода как на эффективных языках программирования. Провести эксперименты сравнения времени выполнения ряда алгоритмов на различных аппаратных средствах.

Научная новизна заключается в следующем:

1. Предложенный метод динамической компиляции Python программ, ориентированных на обработку массивов, значительно повышает быстродействие выполнения кода по сравнению со стандартным интерпретатором языка Python.
2. Разработана программная реализация метода, сочетающего в себе динамичную компиляцию в совокупности с набором оптимизаций разного уровня и реализацией параллелизма данных и применяется для эффективной реализации примитивов и операторов библиотеки расширения массивов NumPy, которая является незаменимым инструментом при выполнении анализа данных и научных вычислений на языке Python.

Практическая ценность заключается в повышении как скорости выполнения программного кода, так и производительности его написания. Метод обеспечивает сочетание удобства использования языка Python и скорости выполнения кода на эффективных языках программирования.

Апробация роботи. Основные положения и результаты работы будут представлены и будут обсуждаться на научной конференции магистрантов и аспирантов «Прикладная математика и компьютинг» ПМК-2018 (Киев, 22-24 марта 2018).

Структура и объем работы. Магистерская диссертация состоит из введения, четырех глав и выводов.

Во введении представлена общая характеристика работы, обоснована актуальность направления исследований, сформулированы цели и задачи исследований, показано научную новизну полученных результатов и практическую ценность работы.

В первом разделе рассмотрены основные средства трансляции и выполнения программного кода, сделан сравнительный анализ компиляции в машинный код и интерпретации байт-кода, описано проблематику реализации алгоритмов с интенсивными вычислениями над массивами в языке Python и сделано краткий обзор самого языка Python.

Во втором разделе выполнен детальный обзор интерпретации кода в языке Python, описаны методы оптимизации программного кода и компиляции, используемые в предложенном методе, рассмотрено использование промежуточного представления кода и описаны технологии с помощью которых реализуется параллелизм данных.

В третьем разделе описывается реализация метода динамической компиляции Python программ, ориентированных на обработку массивов, подробно рассмотрены все этапы преобразования программного кода функции от начального нетипизированного промежуточного представления к непосредственному выполнению машинного кода, описанные реализованы операторы параллелизма данных и работы с массивами.

В четвертом разделе представлены результаты проведенного тестирования метода на наборе алгоритмов с интенсивными вычислениями, выполняющих обработку массивов, сделан анализ улучшения быстродействия в зависимости от выбранного бэкенда выполнения машинного кода и проведен анализ факторов от которых зависит оптимизация выполнения программного кода.

В выводах представлены результаты проведенной работы.

Работа представлена на 82 листах, содержит ссылки на список использованных литературных источников.

Ключевые слова: динамическая компиляция, параллелизм данных, массивно-ориентированное программирование, Python.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ 3

ВСТУП 4

1 АНАЛІЗ МЕТОДІВ ВИКОНАННЯ ПРОГРАМНОГО КОДУ ТА ПРОБЛЕМИ ШВИДКОДІЇ ІНТЕНСИВНИХ ОБЧИСЛЕНЬ У МОВІ ПРОГРАМУВАННЯ PYTHON 6

1.1 Порівняльний аналіз компіляції та інтерпретації програмного коду 6

1.2 Загальний огляд мови програмування Python 8

1.2.1 Особливості мови програмування Python 9

1.2.2 Проблематика швидкодії виконання інтенсивних обчислень на мові програмування Python 11

1.2.3 Бібліотека розширення масивів NumPy 15

1.3 Висновок до першого розділу 16

2 ОПИС ЗАСОБІВ ОПТИМІЗАЦІЇ ТА МОДЕЛІ ДИНАМІЧНОЇ КОМПІЛЯЦІЇ 18

2.1 Динамічна компіляця Python, як модель виконання програмного коду 18

2.1.1 Інтерпретація вихідного коду програми 20

2.1.2 Віртуальна машина Python 22

2.1.3 Компіляція програмного коду 23

2.2 Проміжне представленя програмного коду 26

2.2.1 Процес побудови синтаксичного дерева 29

2.3 Методи оптимізації програмного коду 31

2.3.1 Метод усунення спільних підвиразів 31

2.3.2 SSA 34
2.3.3 Згортка констант 37
2.3.4 Видалення мертвого коду 38
2.4 Програмний паралелізм обробки даних 40
2.4.1 Технологія OpenMP 42
2.4.2 Технологія CUDA 43
2.5 Висновок до другого розділу 45
3 ПРИНЦИПИ ТА ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ РОЗРОБЛЕНОГО МЕТОДУ ДИНАМІЧНОЇ КОМПІЛЯЦІЇ PYTHON ПРОГРАМ, ОРІЄНТОВАНИХ НА ОБРОБКУ МАСИВІВ 46
3.1 Етапи перетворення програмного коду 50
3.2 Реалізовані оператори та опис їх застосування 60
3.2.1 Прості вирази 60
3.2.2 Оператори присвоєння та контролю виконання програмного коду 62
3.2.3 Оператори визначення параметрів масиву 63
3.2.4 Базові оператори роботи з масивами 64
3.2.5 Оператори роботи з пам’яттю 66
3.2.6 Оператори високого рівня 66
3.3 Висновок до третього розділу 69
4 ТЕСТУВАННЯ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ 70
4.1 Застосування розробленого методу для різних алгоритмів 70
4.2 Висновок до четвертого розділу 81
ВИСНОВКИ 82
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 84

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

JIT-компіляція (англ. Just-in-time compilation, компіляція «на льоту») — технологія збільшення продуктивності програмних систем, що використовують байт-код, шляхом компіляції байт-коду в машинний код або в інший формат безпосередньо під час роботи програми.

DCE (англ. dead code elimination) — оптимізація, що видаляє програмний код, виконання якого не впливає на результат програми.

AOT-компіляція (англ. Ahead-of-Time compilation) — попередня компіляція компонентів програми та їх шаблонів під час процесу зборки.

SSA (англ. Static single assignment form) — проміжне представлення, яке використовується компіляторами, в якому для кожної змінної значення присвоюється лише один раз.

Центральний процесор, ЦП (англ. Central processing unit, CPU) — електронний блок або інтегральна схема (мікропроцесор), що виконує машинні інструкції (код програм), головна частина апаратного забезпечення комп'ютера або програмованого логічного контролера.

Графічний прискорювач (анлг. graphic processing unit, GPU) — електронний пристрій, частина комп'ютера, призначена для обробки і генерації зображень з подальшим їхнім виведенням на екран периферійного пристрою.

Обчислення загального призначення на графічних процесорах (англ. General-purpose computing for graphics processing units) — це використання графічного процесора (GPU), який зазвичай обробляє обчислення тільки для комп'ютерної графіки, для того щоб виконанати обчислення в додатках, традиційно виконуваних центральним процесором (CPU).

ВСТУП

Мова програмування Python стала популярною платформою для аналізу даних і наукових обчислень. Python транслює інструкції вихідного програмного коду в проміжне представлення, відоме як байт-код, і потім інтерпретує цей байт-код. Байт-код забезпечує переносимість програм, оскільки це платформо-незалежний формат. Однак через те, що Python не створює двійковий машинний код (наприклад, машинні інструкції для мікропроцесора Intel), деякі програми на мові Python можуть працювати повільніше своїх аналогів, написаних на мовах, що компілюються у машинний код, таких як C. Для вирішення проблеми низької продуктивності стандартного інтерпретатора мови Python, математично інтенсивні обчислення, як правило, переносяться на бібліотечні функції, які написані на високопродуктивних мовах програмування. Якщо така бібліотека для виконання певного алгоритму відсутня, програмісту доводиться прийняти низьку продуктивність або перейти на мову нижчого рівня для ефективної реалізації поставленої задачі.

Таким чином, у процес розробки проекту входить етап прототипування алгоритмів на мови програмування нижчого рівня, а потім безпосередньо відбувається процес переносу розділів з низькою продуктивністю у такій мові як Python, на мову нижчого рівня. Цей етап може займати багато часу, призводити до появи помилок і відводить увагу розробника від початкової задачі.

Отже, існує необхідність у реалізації методу, за допомогою якого буде можливим поєднати зручність та практичність використання мови Python та досягти швидкості виконання коду у алгоритмах з інтенсивними обчисленнями рівня мови низького рівня, що використовує процес компіляції. Це стає можливим за рахунок використання динамічної компіляції, що дозволяє перенести виконання певних ділянок коду, позначених користувачем, з інтерпретатора мови Python на виконання

машинних інструкцій центральним процесором або графічним прискорювачем.

Наукова новизна даної роботи полягає в тому, що динамічна компіляція у сукупності з набором оптимізацій різного рівня та реалізацією паралелізму даних застосовується для ефективної реалізації примітивів та операторів бібліотеки розширення масивів NumPy, яка є незамінним інструментом при виконанні аналізу даних та наукових обчислень мовою Python. Завдяки цьому швидкість виконання алгоритмів з інтесивними обчисленнями над масивами з використанням запропонованого методу навіть перевершує результати мов низького рівня, що застосовують виключно компіляцію для виконання коду та не використовують додаткових оптимізацій.

В даній роботі запропоновано та детально розглянуто метод динамічної компіляції Python програм орієнтованих на обробку масивів, який завдяки динамічній компіляції, оптимизаціям різного рівня, типізації даних та використанню переваг паралельного виконання коду надає значне прискорення часу виконання алгоритмів, які виконують операції над масивами. За рахунок цього розробникам надається потужний інструмент, з використанням якого вдається поєднати переваги програмування мовою Python та використання мов низького рівня у задачах, де необхідна максимальна ефективність виконання коду. При цьому перенесення ділянок програмного коду з виконання інтерпретатором мови Python на компіляцію у машинний код відбувається таким чином, що розробнику не доводиться виконувати цей процес переносу вручну, тож це дозволяє зекономити час роботи над проектом і зосередитись на вирішенні поставленої задачі.

1 АНАЛІЗ МЕТОДІВ ВИКОНАННЯ ПРОГРАМНОГО КОДУ ТА ПРОБЛЕМИ ШВИДКОДІЇ ІНТЕНСИВНИХ ОБЧИСЛЕНЬ У МОВІ ПРОГРАМУВАННЯ PYTHON

1.1 Порівняльний аналіз компіляції та інтерпретації програмного коду

Для виконання програмного коду існують такі інструменти, як компіляція і інтерпретація, за допомогою яких відбувається перетворення програмного коду у форму, зрозумілу комп'ютеру [1 - 4]. Програмний код може бути виконаний нативно в операційній системі після його конвертації в машинний шляхом компіляції, або ж він може виконуватися іншим засобом, який робить це замість операційної системи, а саме інтерпретатором.

Програма, написана мовою програмування, що застосовує компіляцію вихідного програмного коду, у результаті процесу компіляції перетворюється на набір інструкцій цільової машини. Кожна машинна інструкція виконує певну дію, таку як операція з даними, наприклад додавання або копіювання інформації у регістрі або пам'яті, або перехід до іншої ділянки коду, тобто відбувається зміна порядку виконання, при цьому перехід може бути як безумовним, так і умовним, таким, що залежить від результатів попередніх інструкцій. Програмний код після процесу компіляції перетворюється у послідовність таких атомарних машинних операцій.

Мова програмування, що застосовує інтерпретацію, не перетворюється безпосередньо у машинний код для виконання центральним процесором на відміну від мов програмування, що застосовують процес компіляції. Програмний код, написаний такою мовою, виконується за допомогою спеціальної програми-інтерпретатора. По своїй суті інтерпретатор є прошарком програмної логіки між програмним кодом та апаратною частиною комп'ютера. За допомогою

інтерпретатора програмний код поетапно зчитується, обробляється і одразу відбувається виконання інструкцій.

Головною перевагою компіляції є швидкість виконання програмного коду. Так як програмний код конвертується у машинний код, програми, що компілюються, виконуються набагато швидше та ефективніше ніж ті, що інтерпретуються, особливо якщо врахувати складність конструкцій сучасних інтерпретуємих мов, таких як Python.

Низькорівневі мови як правило застосовують компіляцію, так як зазвичай фактор ефективності у них ставиться вище за кросплатформеність. Крім того, компіляція надає розробнику набаго більше можливостей у плані контролю апаратного забезпечення, наприклад управлінням пам'яттю та використанням процесору.

Проблеми компіляції програмного коду є досить очевидними. Для запуску програми, написаній на мові програмування, що застосовує компіляцію, її спочатку необхідно безпосередньо скомпілювати. Це є не лише зайвим кроком, на якому втрачається інколи багато часу, а ще й значним ускладненням відлагодження написанної програми, адже для тестування будь-якої зміни програму необхідно компілювати знову. Крім того, мови, що компілюються, є платформо-залежними, оскільки машинний код залежить від машини, на якій компілюється і виконується програма.

На відміну від мов програмування, що компілюють програмний код, інтерпретація не потребує для виконання машинний код, замість нього програму рядок за рядком виконують інтерпретатори. Раніше процес інтерпретації займав дуже багато часу, але з приходом таких технологій, як динамічна компіляція, розрив між компіляцією та інтерпретацією значно зменшився. Головними перевагами інтерпретації є:

- незалежність від платформи;
- рефлексія;

- динамічна типізація;
- менший розмір виконуваних файлів;
- динамічні області видимості.

Головним недоліком мов програмування, що застосовують інтерпретацію програмного коду є їх невисока швидкість виконання. Проте динамічна компіляція дозволяє прискорити процес виконання коду за рахунок переводу окремих частин програми у машинний код.

1.2 Загальний огляд мови програмування Python

Мова програмування Python з'явилася в 1991 році, заявивши про себе як про мультипарадигмальну мову, це означає, що програми пишуться на одній мові, але можуть бути написані в різних стилях. Також вона є об'єктно-орієнтованою (працює з полями і методами), багатоплатформною (на Python можна з однаковим набором можливостей програмувати як на операційній системі Windows, так і на MacOS, Linux, *nix та інших популярних операційних системах). Мова Python є рефлективною, тобто програма може аналізувати свою структуру і змінювати її по мірі виконання коду; імперативною (відбувається виконання прямих інструкцій, «наказів»); функціональною (підтримує символьну обробку даних) та аспектно-орієнтованою, програма поділяється на модулі-аспекти.

Тобто, маючи мінімалістичний синтаксис, мова Python не поступається, а іноді і перевершує в потужності більш складні середовища програмування. Цей мінімалізм дозволяє підвищити швидкість написання програм, а також підвищує простоту читаємості коду. При цьому стандартна бібліотека модулів включає в себе постійно набір різних функцій, що постийно збільшується, а відсутні користувач може досить легко вбудувати сам.

1.2.1 Особливості мови програмування Python

В чіткій, але досить простій структурованості коду полягає причина популярності мови Python. Відсутність громіздких конструкцій, що позначають нові класи, методи або цикли, замінюється виділенням пробілами і табуляцією, що є більш наочним способом. В таких умовах набагато легше відстежувати прогрес створення програми, її легше налагоджувати і доповнювати. Структурований Python код значно простіше зрозуміти як новачку в програмуванні, так і фахівцю, який змінив свою основну мову програмування. При цьому справедливо буде зауважити, що без наявності хоча б теоретичних знань про об'єктно-орієнтоване програмування, знайомство з Python та іншими мовами буде ускладнено.

Проте красивого оформлення недостатньо для досягнення успіху в умовах жорсткої конкуренції між розробниками різних мов програмування. Нижче наведені інші вирішальні фактори, що є критеріями успіху Python на світовій арені [5]:

- Інтерпретуємість мови. По аналогії роботи з мовами Lisp і Prolog, користувачеві доступний набір різних інтерпретаторів. Вони являють собою графічні інтерфейси мови програмування, що спрощують роботу з нею. Наприклад, в стандартному дистрибутиві Python вже є інтерпретатор, який виконує кожну введену програмістом команду. Таким чином можна сильно скоротити часові витрати на проект, коли для перевірки певної ділянки коду не потрібно складати повноцінну програму, а можна відразу побачити результат дії тієї чи іншої функції.
- Об'єктно-орієнтований підхід. ООП - основна модель, на основі якої реалізована мова Python, проте дана модель дещо відрізняється від традиційної: класи можуть бути і є об'єктами

всередині самої програми; підтримується множинне наслідування; присутній віртуальний поліморфізм класів; наявна інкапсуляція на всіх рівнях; присутність конструкторів, деструкторів і прибиральників сміття в базовій збірці; розвинені емулятивні можливості; наявні вбудовані методи для роботи з найбільш повторюваними операціями; підтримується метапрограмування; методи, поля і класи є статичними.

- Функціональность програмування. Можлива робота з функцією як об'єктом, наявна повноцінна рекурсивность, замикання, звернення до частин функцій і можливість створення власних засобів.
- Пакетно-модульна система розширень мови. Python, завдяки своїй простоті, має безліч надбудов і доповнень, за допомогою яких є можливість гнучко збільшити можливості мови по мірі необхідності. Підключення нових модулів виконується в одну команду в коді або інтерпретаторі.
- Інтроспекція. Ця функція мови дозволяє за запитом одержувати детальну інформацію про будь-який об'єкт, який може знаходитися всередині програми, що виконується.

Ці та багато інших функцій вже є частиною мови і там, де для багатьох більш складних середовищ потрібні додаткові модулі, Python справляється за допомогою свого більш компактного базового дистрибутиву. Якщо ж необхідної надбудови немає, її встановлення або самостійне написання не потребує багато часу.

У порівнянні з мовами, що компілюються або є строго типізованими, такими як C, C++ і Java, Python у багато разів підвищує продуктивність праці розробника. Обсяг програмного коду на мові Python зазвичай становить третину або навіть п'яту частину еквівалентного програмного коду на мові C++ або Java. Це означає менший обсяг введення з

клавіатури, менша кількість часу на налагодження і менший обсяг витрат часу на підтримку. Крім того, програми на мові Python запускаються відразу ж, минаючи тривалі етапи компіляції, що необхідні в деяких інших мовах програмування, за рахунок цього продуктивність праці програміста зростає ще більше.

1.2.2 Проблематика швидкодії виконання інтенсивних обчислень на мові програмування Python

Мова програмування Python зарекомендувала себе як популярне середовище для складних логічних розрахунків та аналізу даних. На відміну від Mathematica, Matlab або R, Python на етапі створення не була задумана як математична, числова або статистична мова програмування. Однак, в силу семантичної гнучкості мови Python та зручності взаємодії з мовами нижчого рівня, Python отримала надзвичайно багату екосистему наукових бібліотек. До них відносяться такі бібліотеки, як наприклад NumPy і Pandas, які надають велику кількість числових структур даних, SciPy, великий репозиторій математичних функцій, а також matplotlib, гнучка бібліотека для побудови графіків. Універсальність цих інструментів, а також допомога спільноти розробникив, які розробляють та використовують їх, призвели до того, що Python стає однією з основних мов програмування у ряді наукових дисциплін.

В сучасній реалізації Python компілює (тобто транслює) інструкції вихідного програмного коду в проміжне представлення, відоме як байт-код, і потім інтерпретує цей байт-код. Байт-код забезпечує переносимість програм, оскільки це платформонезалежний формат. Однак через те, що Python не створює двійковий машинний код (наприклад, машинні інструкції для мікропроцесора Intel), деякі програми на мові Python можуть працювати повільніше своїх аналогів, написаних на компілюємих мовах, таких як C. Помітність різниці в швидкості виконання програм, залежить від того, якого роду програми пишеться. Python багаторазово піддавався

оптимізації і в окремих прикладних областях, програмний код на цій мові відрізняється досить високою швидкістю виконання.

Крім того, коли в сценарії Python відбувається який-небудь значний процес, наприклад обробляється файл або конструюється графічний інтерфейс, програма фактично виконується зі швидкістю, яку здатна надати мова С, тому що задачі такого роду вирішуються скомпільованним з мови С програмним кодом, що лежить в ядрі інтерпретатора Python. Набагато важливіше те, що перевага в швидкості розробки часом важливіша за втрату швидкості виконання коду, особливо якщо врахувати швидкодію сучасних комп'ютерів.

Проте навіть при високій швидкодії сучасних процесорів залишаються такі галузі, де потрібна максимальна швидкість виконання. Реалізація математичних обчислень і анімаційних ефектів, наприклад, часто вимагає наявності базових обчислювальних компонентів, які вирішують свої завдання зі швидкістю мови С (або ще швидше).

Якщо за мету поставлено швидке виконення деяких повторюваних арифметичних операцій на великому наборі чисел, то в ідеалі ці числа мають бути збережені в пам'яті на постійній основі, та завантажуватися в регістри невеликими групами, а операції над ними потрібно виконувати за допомогою невеликого переліку машинних інструкцій. Інтерпретатор Python натомість представляє чисельні значення використовуючи велику кількість виділених об'єктів і навіть прості операції, такі як перемноження двох чисел, реалізуються за допомогою ресурсозатратних засобів.

Якщо низька продуктивність інтерпретатора Python не могла бути яким-небудь чином усунена, то розробникам, які користуються математичними засобами мови Python, довелося би прийняти низьку продуктивність виконання коду. Ключове вирішення проблеми задоволення потреб високої ефективності виконання програмного коду полягає в тому, щоби перенести обчислювально інтенсивні задачі на мови нижчого рівня, за допомогою необхідних бібліотек. Якщо алгоритм

більшість свого часу роботи виконується за допомогою ефективної бібліотеки, тоді не повинно бути значної різниці у продуктивності між реалізацією на мові Python та еквівалентною програмою, що використовує ті ж самі бібліотекі з більш ефективної мови програмування.

Бібліотека для роботи з масивами NumPy [6 - 7] відіграє важливу роль у тому, щоб надати можливість використовувати ефективну алгоритмічну реалізацію операцій над масивами. Ці масиви можуть легко переноситися на попередньо скомпільований код C або Fortran. Масив NumPy являє собою чисельний контейнер для програм Python, за допомогою якого звичайно будуються інші структури даних.

Розробка за допомогою бібліотеки NumPy надає прийнятну продуктивність виконання коду до того моменту, поки алгоритм, який необхідно реалізувати, може бути вираженим за допомогою існуючих скомпільованих примітивів, проте розробнику може знадобитися використати засоби, для яких не існує попередньо скомпільованих аналогів. У цьому випадку розробнику необхідно здійснити вибір: або розробити необхідне обчислення на чистій мові Python, при цьому змиритися з значною втратою ефективності виконання програмного коду; або розробити дане обчислення на мові нижчого рівня і зіштовхнутися з значною втратою продуктивності написання програмного коду. Розробникам бібліотеки scikit-learn, яка широко використовується для задач машинного навчання, довелося витратити більш ніж половину року для розробки і перегляду статично скомпільованого дерева вибору рішень. Версія розроблена виключно мовою Python була би значно корочшою та простішою, проте не мала би жодного шансу на досягнення необхідного рівня продуктівності виконання.

Проте навіть просте перенесення певних алгоритмів на мови нижчого рівня не надасть доступ до повного використання обчислювальних ресурсів. Сучасній стаціонарний комп'ютер зазвичай буде мати процесор з 2-8 ядрами та графічний процесор (GPU), здатний на

обчислення загального призначення. GPU, завдяки своїй архітектурі, що полягає у поєднанні багатьох ядер для розпаралелення виконання задач, в залежності від задачі може бути у 10-100 разів швидший за реалізацію на багатоядерному центральному процесорі (CPU). Деякі області досліджень, такі як глибокі нейронні мережі, стали цілком залежними від обчислювальної потужності GPU для виконання завдань, які є неприпустимо повільними при виконанні на CPU. На жаль, використання паралельного апаратного забезпечення не є простим. Розподілення обчислень на декількох ядрах вимагає використовувати бібліотеку для роботи з потоками або паралельний API, такий як OpenMP [10]. Прискорення на GPU вимагає ще більше зусиль від програміста, необхідні глибокі знання графічних прискорювачів та використання спеціалізованої мови, такої як CUDA або OpenCL.

Окрім двох шляхів розробки, один з яких є написанням з використанням виключно мови програмування Python та втратою від цього ефективності виконання коду, другим є шлях з використанням вставок з мов програмування нижчого рівня при якому втрачається продуктивність написання коду, існує ще один спосіб, який полягає у динамічній компіляції. Загальні проблеми втрати ефективності виконання коду при розробці мовою Python можуть бути усунені, якщо машинний код буде генеруватися під час виконання програми, коли буде вказана необхідна інформація для спеціалізованої компіляції.

Існує можливість використовувати алгоритми описані мовою високого рівня, які можна дістати з такої мови як Python, для створення коду який швидший ніж реалізація на C або Fortran. Розглянемо декілька факторів, які сприяють значному прискоренню відносно написаного вручну коду низького рівня: попередньо скомпільовані примітиви мають бути скомпільовані окремо і тому вони не можуть виконувати оптимізацію, коли ці процедури використовуються разом одночасно. Таким чином, при композиції декількох таких функцій можуть виникнуті

марні витрати обчислювальних ресурсів. Наприклад при виконанні операцій обчислення можуть створюватися тимчасові значення, яких можна уникнути, при цьому уникнувши додаткових виділень або переміщень даних.

Реалізація чутливої до ефективності виконання частини програмного коду алгоритму на мові C або Fortran сама по собі не робить нічого для того, щоб використати паралельне апаратне забезпечення, яке є в більшості сучасних комп'ютерів. Якщо, наприклад, розробник хоче перемістити деякі обчислення на свій графічний процесор, йому необхідно навчитися незнайомій і складній моделі програмування, такій як OpenCL або CUDA. Якщо існує надія на те, що програмування обчислень загального призначення за допомогою графічного процесора стане легкою та доступною діяльністю, це ймовірно відбудеться завдяки процесу автоматичному переносу програм високого рівня на графічний процесор.

1.2.3 Бібліотека розширення масивів NumPy

Бібліотека для мови програмування Python NumPy є бібліотекою з відкритим вихідним кодом. Можливості, запропоновані бібліотекою:

- підтримка багатовимірних масивів (включаючи матриці);
- підтримка високорівневих математичних функцій, призначених для роботи з багатовимірними масивами.

Математичні алгоритми, реалізовані на інтерпретованих мовах (наприклад, як Python), часто працюють набагато повільніше тих же алгоритмів, реалізованих на компільованих мовах (наприклад, Fortran, C, Java). Бібліотека NumPy надає реалізації обчислювальних алгоритмів (у вигляді функцій і операторів), що оптимізовані для роботи з багатовимірними масивами. В результаті будь-який алгоритм, який може бути виражений у вигляді послідовності операцій над масивами

(матрицями) і реалізований з використанням NumPy, працює так само швидко, як еквівалентний код, що виконується наприклад в MATLAB.

Бібліотека NumPy пропонує широку функціональність, та надає можливість для реалізації:

- роботи з багатовимірними масивами;
- перетворення списків і кортежів Python в масиви NumPy (і навпаки);
- імпорту даних з текстових файлів;
- математичних функцій (арифметичних, тригонометричних, експоненціальних, логарифмічних тощо);
- розподілу ймовірностей;
- статистичних параметрів (середнє, стандартне відхилення, гістограми);
- перетворення Фур'є;
- застосовування лінійної алгебри;
- записів даних в текстові файли;
- інтеграції в існуючі програми на Python;

Перевага NumPy над іншими пакетами для наукових обчислень з подібними ліцензіями (такими як GNU Octave), полягає в тому, що є можливість створювати додатки на мові програмування Python, що використовують можливості NumPy і всіх інших пакетів Python, забезпечуючи величезним асортиментом інструментів для вирішення найрізноманітніших задач. Крім того, синтаксис NumPy успадкований від мови Python, що дозволяє відмовитися від синтаксису в стилі MATLAB і застосовувати свої навички програмування на Python.

1.3 Висновок до першого розділу

У першому розділі проведено порівняльний аналіз існуючих методів виконання програмного коду, а саме інтерпретації та компіляції. Мови програмування, що застосовують компіляцію, є найбільш ефективними у

плані швидкодії виконання програмного коду, оскільки вони виконуються як машинний код і дозволяють використовувати апаратне забезпечення системи. Однак це вводить додаткові обмеження на написання коду і робить його залежним від платформи. Мови програмування, що застосовують інтерпретацію, не залежать від платформи і дозволяють використовувати такі техніки динамічного програмування, як метапрограмування. Також вони мають ряд переваг у порівнянні з мовами, що компілюються. Проте, в швидкості виконання програмного коду вони значно поступаються, але цей недолік може бути усуненим за рахунок використання динамічної компіляції.

Популярною на сьогоднішній день мовою програмування серед великої кількості запропонованих мов є мова Python. Таку високу позицію для програмістів в рейтингу мов, що слугують для вирішення задач мова Python посідає з ряду причин: простота програмного коду, відсутність складних конструкцій, об'єктно-орієнтований підхід, інтерпретуємість мови та інших. Завдяки перерахованим перевагам відбувається значна економія часу, що витрачається загалом на виконання проекту.

Мова Python може стати надійним інструментом у руках програмістів, робота яких підпорядковується науковим дослідженням та складним математичним обчисленням над масивами. Незамінним інструментом для виконання таких задач є бібліотека розширення масивів NumPy.

Також у даному розділі розглянуто проблему низької швидкодії виконання програмного коду з інтенсивними обчисленнями у мові Python.

2 ОПИС ЗАСОБІВ ОПТИМІЗАЦІЇ ТА МОДЕЛІ ДИНАМІЧНОЇ КОМПІЛЯЦІЇ

2.1 Динамічна компіляця Python, як модель виконання програмного коду

Виконання байт-коду [8 - 9] програм під керівництвом віртуальної машини демонструє ряд переваг перед традиційним виконанням машинного коду. До переваг можна віднести крос-платформеність, безпеку, зручність за рахунок відсутності процесу компіляції та зручність відладки. У той же час, таке виконання тягне за собою втрату швидкості виконання. Тож варто детальніше розглянути два основних методи виконання коду: інтерпретацію та компіляцію. А також для кожного із цих методів зауважити витрати та вузькі місця з точки зору продуткивності.

Віртуальна машина являє собою архітектурну обчислювальну машину, що не залежить від реальної чи апаратної платформи. Аналогічно до реальної обчислювальної машини, віртуальна машина здатна визначити архітектуру команд, яка включає в себе згоду про передачу аргументів, стеки операндів або набір регістрів, модель пам’яті, безпосередньо набір команд. Машинна команда (машинна інструкція) – закодована команда для процесору на виконання певної події стосовно обробки інформації. Машинна інструкція може містити у собі:

- код виконання операції;
- операнди та/або їх адреси;
- указання про розміщення результату;
- посилання на наступну команду.

Етапи виконання машинної інструкції [10] подано на рисунку 2.1.

Рис 2.1 - Етапи формування машинної інструкції

Також віртуальна машина визначає виконавче представлення програми. Прикладом такого представлення може слугувати закодована послідовність інструкцій, аналогічна до машинного коду. Таке представлення називається байт-кодом. По своїй суті байткод є стек-орієнтованою мовою, схожою за своєю структурою на мову асемблер. Щоб зробити операції з даними їх спочатку потрібно покласти на стек. В байткоді немає імен змінних, у них є номери. Нульовий номер у посиланні робить посилання на поточний об'єкт або змінну this. Потім йдуть параметри виконавчого методу, за ними інші змінні.

Порівняно з машинним кодом – байткод є компактнішим, так як набір інструкцій віртульної машини більш високорівневий. Навідміну від інших представлень текста або синтаксичних дерев, байткод простий у декоруванні. Це спрощує його ефективність виконання. Байткод використовується в якості виконавчого представлення в популярних на сьогоднішній день мовах програмування.

Виконання проміжного представлення, без сумнівів, тягне за собою допоміжні накладні витрати у порівнянні з виконанням машинного коду. Але, завдяки виконанню частини коду шляхом його компіляції, застосуванню ряду оптимізацій, сучасні віртуальні машини за швидкістю виконання близько наблизились до швидкості виконання машинного коду.

2.1.1 Інтерпретація вихідного коду програми

Найпростіший та найочевидніший спосіб виконання байткоду заключається в його послідовній інтерпретації (рис. 2.2). Інтерпретатори прості у використанні та крос-платформленні. Інтерпретація не потребує ніякої попередньої роботи перед виконанням програми. Завдяки цим перевага інтерпретація широко використовується для використання байткоду. Головний недолік цього способу – невисока швидкість виконання коду, яка менша швидкості виконання машинного коду [11].

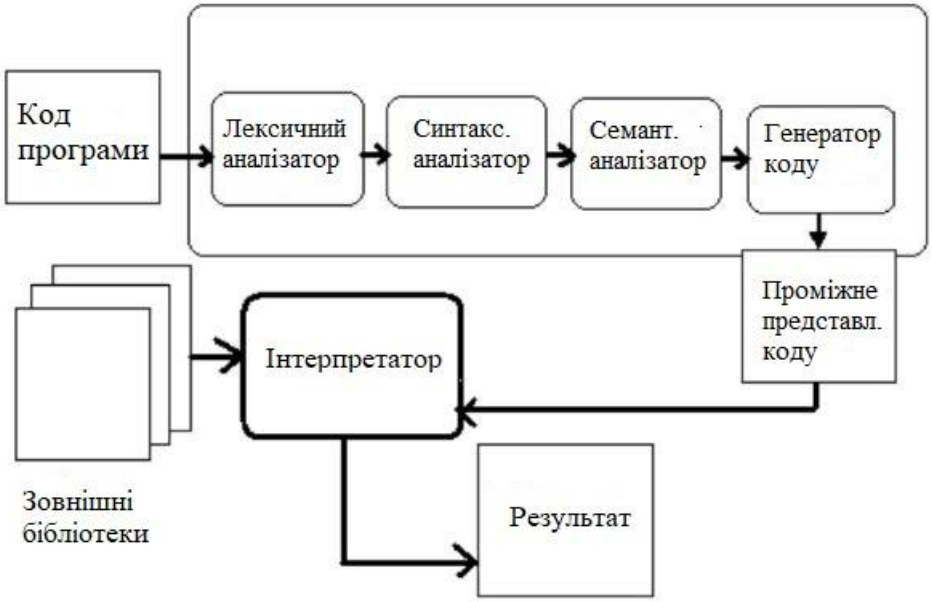


Рис 2.2 - Інтерпретація програмного коду

Для того, щоб виявити причини низької продуктивності, розглянемо роботу інтерпретатора докладніше. Інтерпретація інструкції віртуальної машини складається з декодування, безпосереднього виконання інструкції та переходу до наступної інструкції. Безпосереднє виконання інструкції відбувається лише на другому кроці, решта дій – вимушені накладні витрати на інтерпретацію. Продуктивність інтерпретатора визначається відношенням корисної роботи та цих самих накладних витрат. Це співвідношення варіюється від реалізації інтерпретатора та архітектури набору інструкцій. Очевидно, що це співвідношення можно покращити,

збільшивши кількість роботи, що виконує одна інструкція, тобто використовуючи більш високорівневий набір інструкцій. На декодування та перехід між інструкціями витрачається великий проміжок часу роботи інтерпретатора. Це можна пояснити тим, що більшість реалізацій використовує непрямі переходи (переходи по значенню, визначеному на етапі виконання) при обробці інструкцій. Конвеєрні архітектури сучасних процесорів максимально ефективно виконують послідовні ділянки коду, зміни ж ходу виконання програми приводять до збросу конвеєра та значній втраті швидкості виконання. Для того, щоб уникнути таких ситуацій процесори використовують системи передбачення переходів. Правильно передбачений перехід дозволить продовжити виконання без будь-яких наслідків. Нажаль, системи передбачень переходів майже не працюють для непрямих переходів. Таким чином, обробка кожної інструкції майже завжди призводить до сбросу конвеєра та втраті продуктивності.

Про мову Python говорять в основному як про мову програмування. Але в поточній реалізації це ще і програмний пакет, який називається інтерпретатором. Як уже описувалося раніше інтерпретатор - це такий модуль, який виконує інші програми. Коли ви пишете код на мові Python, інтерпретатор Python читає вашу програму і виконує складові її інструкції. По своїй суті інтерпретатор - це шар програмної логіки між вашим програмним кодом і апаратурою вашого комп'ютера.

В процесі установки пакета Python на комп'ютер створюється ряд програмних компонентів - як мінімум, інтерпретатор і бібліотека підтримки. Залежно від особливостей використання інтерпретатор Python може мати вигляд виконуваної програми або набору бібліотек, пов'язаних з іншою програмою. Залежно від версії Python сам інтерпретатор може бути реалізований як програма на мові C, як набір класів Java або в будь-якому іншому вигляді. Незалежно від різновиду Python програмний код на цій мові завжди буде виконуватися інтерпретатором.

Програміст пише програмний код в текстовий файл, а потім запускає цей файл за допомогою інтерпретатора. Коли інтерпретатор Python отримує від програміста команду запустити сценарій, він виконує кілька проміжних дій, перш ніж ваш програмний код почне виконуватися. Зокрема, сценарій спочатку буде скомпільовано в байт-код, а потім він буде переданий віртуальній машині.

Інтерпретатор зберігає байт-код для прискорення запуску програм. Наступного разу, коли відбудеться запуск програми, Python завантажить файл .pyc і мине етап компіляції - за умови, що вихідний текст програми не змінювався з моменту останньої компіляції. Щоб визначити, чи потрібно виконувати перекомпіляцію, Python автоматично порівнює час останньої зміни файлу з вихідним текстом і файлу з байт-кодом. Якщо вихідний текст зберігався на диск після компіляції, при наступному його запуску інтерпретатор автоматично виконає повторну компіляцію програми. Якщо інтерпретатор не в змозі записати файл з байт-кодом на диск, програма від цього не постраждає, просто байткод буде згенеровано в пам'яті і зникне після закінчення программи. Однак оскільки файли Python підвищують швидкість запуску програми, може знадобитися мати можливість зберігати їх, особливо для великих програм. Крім того, файли з байткодом - це ще один із способів поширення програм на мові Python.

2.1.2 Віртуальна машина Python

Як тільки програма буде перетворена в байт-код (або байт-код буде завантажений з існуючих файлів), він передається механізму під назвою віртуальна машина Python (англ. PVM) [12 - 13]. Фактично PVM - це просто великий цикл, який виконує перебір інструкцій в байт-коді, одна за одною, і виконує відповідні їм операції. PVM - це механізм часу виконання, що завжди присутня в складі системи Python і це той самий програмний компонент, який виконує сценарії. Формально - це остання складова інтерпретатору Python. Компіляція в байт-код виконується

автоматично, а PVM - це всього лише частина системи Python, яку встановлюють на комп'ютер. На рис 2.3 зображено злагоджену роботу інтерпретатора Python.

Рис 2.3 - Інтерпретатор Python

2.1.3 Компіляція програмного коду

На відміну від інтерпретації, скомпільований код не виконує зайвої роботи по декоруванню та переходам між інструкціями байткоду. Крім того, скомпільований код може бути додатково оптимізований. Можливості компілятора у цьому плані значно ширші, ніж у інтерпретора, що виконує інструкції у відриві від контексту. Наприклад, компілятор може більш ефективно розподілювати регістри в межах методів, володіючи інформацією про локальні змінні та проміжні значення, що використовуються. Якщо компілятор віртуальної машини працює одночасно з виконанням програми, то він може використовувати інформацію про динамічний стан програми, а ткож може реагувати на зміни цього стану.

З іншої сторони, компіляція також має і свої недоліки. Компілятор значно ускладнює архітектуру віртуальної машини та погіршує її переносимість, так як використовує інформацію про цільовий процесор. Компілятор потребує більше пам’яті у порівнянні з інтерпретатором для зберігання внутрішніх структур скопмільованого коду, який зазвичай в декілька разів більший за розмір відповідного байткоду. На відміну від інтерпретатору компілятор не може одразу перейти до виконання програми: завжди існує затримка, пов’язана з компіляцією в машинний код. Якщо компіляція здійснюється на льоту, то компілятор конкурує з

виконавчою програмою за ресурси процесору. Через все це виникають ускладнення застосування компіляції в умовах обмежених ресурсів.

Можна виділити дві стратегії компіляції байткоду в код цільової платформи: компіляція перед виконанням (англ. AOT) та динамічна компіляція (англ. JIT). В першому підході AOT компілятор виступає в ролі статичного бекенд компілятору, котрий транслює машинно-незалежне проміжне представлення в бінарний зразок [14 - 15]. Для цього застосовуються добре відомі алгоритми статичної компіляції. Як і статичний компілятор, AOT компілятор не обмеженний у часі та може виконувати складні оптимізації, що потребують глибокого аналізу програми.

Компіляція програми одночасно з її виконанням називається JIT компіляцією або динамічною компіляцією. Так як динамічний компілятор працює одночасно з програмою, то він конкурує з програмою за ресурси системи, а час його роботи стає критичним для швидкості роботи програми. Цей випадок дуже впливає на алгоритми та оптимізації, що застосовуються при динамічній компіляції. Якщо статичний компілятор має можливість виконувати більш складні оптимізації ціною збільшенням часу компіляції, то динамічний компілятор повинен балансувати між якістю породжуваного коду та часом роботи для досягнення оптимальної продуктивності. У цей же час, динамічний компілятор володіє додатковими знаннями про статистику виконання програми, її динамічний стан та поточну апаратну платформу. Це дає додаткові можливості для оптимізації, що недоступні для статичного компілятору [16 - 17].

Система Psyco - це не інша реалізація мови Python, а компонент, який розширює модель виконання байт-коду, що дозволяє програмам виконуватися швидше [18]. Psyco - це розширення PVM, яке збирає і використовує інформацію про типи, щоб транслювати частини байт-коду програми в істинний двійковий машинний код, який виконується набагато

швидше. Для такої трансляції не потрібно вносити зміни в початковий програмний код або виробляти додаткову компіляцію в ході розробки.

Якщо бути точним, під час виконання програми Psyco збирає інформацію про типи об'єктів і потім ця інформація використовується для генерації високоефективного машинного коду, оптимізованого для об'єктів цього типу. Після цього вироблений машинний код заміщає відповідні ділянки байт-коду і тим самим збільшує швидкість виконання програми. В результаті при використанні Psyco істотно зменшується загальний час виконання програми. В ідеалі деякі ділянки програмного коду під управлінням Psyco можуть виконуватися так само швидко, як скомпільований код мови С. Оскільки ця компіляція з байт-коду проводиться під час виконання програми, зазвичай Psyco називають динамічним компілятором. Однак насправді Psyco трохи відрізняється від JIT компіляторов. Насправді Psyco - це спеціалізований JIT-компілятор; він генерує машинний код, оптимізований для типів даних, які фактично використовуються в програмі. Наприклад, якщо одна і та ж ділянка програми використовує різні типи даних в різний час, Psyco може генерувати різні версії машинного коду для підтримки кожної з комбінацій.

Застосування Psyco показує суттєве збільшення швидкості виконання програмного коду Python. Згідно з інформацією, яка подана на домашній сторінці проекту, Psyco забезпечує збільшення швидкості «від 2 до 100 разів, зазвичай в 4 рази, при використанні немодифікавоного інтерпретатора Python, незмінного початкового тексту, всього лише за рахунок використання динамічно завантаженого модуля з розширення на мові С». За інших рівних умов найбільший приріст швидкості спостерігається для програмного коду, що реалізує різні алгоритми чистою мовою Python, - саме такий програмний код зазвичай переносять на мову С з метою оптимізації.

Профілювальник являється обов’язковою частиною механізму динамічної компіляції. Профілювання не повинне сильно уповільнювати виконання програми.

Відповідальним за вибір JIT-компілятора й інтерпретатора у мові Python є менеджер виконання (execution manager). Він засновує свій вибір на конфігурації віртуальної машини і профілі, зібраному під час виконання.

У режимі інтерпретації менеджер виконання відправляє всі методи на виконання до інтерпретатора.

У режимі JIT-компіляції менеджер виконання робить наступне:

- запускає і конфігурує профілювальники
- конфігурується JIT-компілятори так, щоб забезпечити можливість використовувати дані, отримані в результаті профілювання
- визначає логіку перекомпіляції

Таким чином, робота віртуальної машини і компіляторів повністю конфігурується, дозволяє збирати і використовувати профіль, а також забезпечує можливість динамічної оптимізації шляхом перекомпіляції.

Стандартна серверна конфігурація роботи віртуальної машини - ланцюжок з двох компіляторів, де перший компілятор збирає профіль, а другий має набагато більший конвеєр оптимізацій і компілює тільки «гарячі» методи, тобто методи, що викликаються досить велике число раз (задається в конфігурації).

2.2 Проміжне представленя програмного коду

Спершу функція, яка обернена декоратором, представляється у вигляді проміжного представлення, яке являю собою синтаксичне дерево, але без уточнення типів даних. Це представлення грає роль не типізованого

шаблону функції, яке потім використовується при виклику функції з наборами аргументів певного типу.

Для проміжного представлення використовується синтаксичне дерево, де тіло кожної функції являє собою набір певних виразів. Оператори, які відповідають за цикли та окремі гілки коду можуть мати свої внутрішні набори виразів. Хід виконання програми має бути чітко структурованим, це означає, що неможливо використати довільні переході між різними частинами програми.

Двома головними трансформаціями які відбуваються при перетворенні коду з синтаксису мови Python до проміжного представлення є:

- SSA конверсія. SSA (англ. Static single assignment form) - проміжне представлення, яке використовується компіляторами, в якому для кожної змінної значення присвоюється лише один раз. Змінні вихідної програми розбиваються на версії, зазвичай за допомогою додавання суфікса, таким чином, що кожне присвоювання здійснюється на унікальній версії змінної.
- Лямбда підйом. Кожний виклик функції, який знаходиться всередині іншої функції, піднімається на рівень вище за рахунок того, що локальні змінні які використовуються у вкладеній функції передаються до неї як параметри, за рахунок цього стає можливим передавати аргументи у відповідні регістри, а не у стек.

Початкова стадія компілятора будує проміжне представлення результуючої програми, на підставі якого завершальна стадія генерує цільову програму.

Два види проміжних представлень:

- дерева, включаючи дерева розбору і (абстрактні) синтаксичні дерева
- лінійні представлення, зокрема трьох-адресний код

У процесі синтаксичного апалізу для представленя важливих програмних конструкцій створюються вузли синтаксного дерева. У процесі аналізу до них додається різна інформація у вигляді атрибутів, пов'язаних з цими вузлами. Вибір атрибутів залежить від трансляції, що виконується.

Трьох-адресний код, в свою чергу, являє собою послідовність елементарних програмних кроків, таких як, наприклад, складання двох величин. На відміну від дерев, тут немає ієрархічної структури, таке уявлення стає необхідним, якщо ми маємо намір виконати істотну оптимізацію коду. В цьому випадку ми розбиваємо довгу послідовність трьох-адресних інструкцій, що розділить програму, на базові блоки, які являють собою послідовності інструкцій, що завжди виконуються одна за одною, без розгалуження. Така перевірка називається синтаксичною; в загальному випадку «синтаксичний» означає «той, що виконується компілятором» . Статична перевірка гарантує, що певні види програмних помилок, включаючи невідповідність типів, виявиться під час компіляції.

Компілятор може будувати синтаксичне дерево одночасно із генерацією трьох-адресного коду. Однак зазвичай компілятори генерують трьох-адресний код під час того, коли синтаксичний аналізатор будує дерево, але без явної побудови завершенного синтаксичного дерева.

Окрім створення проміжного представлення, на початковій стадії перевіряється, що вихідна програма відповідає синтаксичним і семантичним правилам мови програмування, якою написаний код. Замість цього компілятор зберігає вузли та її атрибути, необхідні для семантичних перевірок або інших задач, разом із структурами даних, що необхідні для синтаксичного аналізу. При цьому частини синтаксичного дерева, що потрібні для побудови трьох-адресного коду, виявляються доступними в

той момент, коли це необхідно, але, коли потреби в них не має, вони знищуються.

2.2.1 Процес побудови синтаксичного дерева

Спочатку наведемо схему трансляції для побудови синтаксичного дерева на рисунку 2.4.

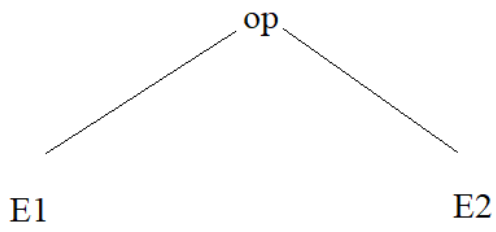


Рис. 2.4 - Схема трансцяції коду

Синтаксичне дерево представляє собою вираз, сформульоване застосуванням оператора op до підвиразу, що представлений вузлами E1 та E2. Синтаксичні дерева можуть бути побудовані для будь-якої конструкції, а не лише для виразів. Кожна конструкція, що представлена вузлом, прилеглі вузли якого представляють семантично значимі компоненти конструкції. Наприклад, семантичне значення компонента циклу while мови програмування C - while (expr) stmt являються вирази expr та інструкція stmt. Вузол синтаксичного дерева для такої конструкції містить оператор, який ми називаємо while, і має два прилеглих вузла – синтаксичні дерева для expr та stmt.

Частиною синтаксичних дерев є блоки (рис. 2.5). Розглянемо правила, що відносяться до блоків:

stmt $\longrightarrow$ block {stmt.n = block.n}

block $\longrightarrow$ ‘{ stmts ‘}’ {block.n = stmts.n}

Перше правило каже, що якщо інструкція являє собою блок, то вона має те саме синтаксичне дерево, що й блок. Друге правило – синтаксичне дерево для нетерміналу block являє собою просто синтаксичне дерево для послідовності інструкцій у блоці. Таким чином, блоки, з оголошенням та

без них, в проміжному коді представляють собою всього лиш ще один різновид інструкції.

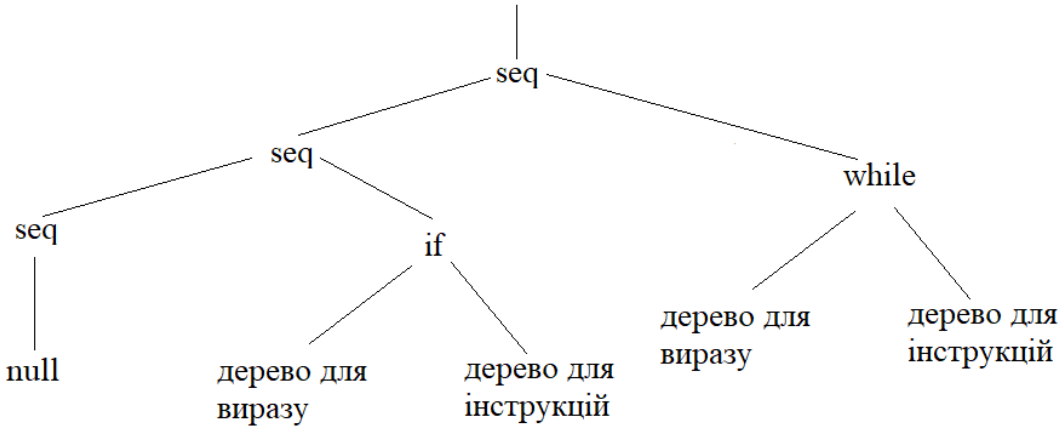


Рис 2.5 - Частина синтаксичного дерева для списку інструкцій, що складаються з інструкцій if та while

Синтаксичні перевірки представляють собою перевірки узгодженості, що виконуються у процесі компіляції. Вони не тільки гарантують, що програма буде успішно скомпільована, а й мають потенціал для ранього перехвату програмних помилок – до виконання програми. Статичні перевірки містять у собі наступне:

- Перевірки синтаксису. Ці перевірки мають більше відношення до синтаксису, ніж до граматики. Наприклад, перевірки виконання таких обмежень, як те, що ідентифікатор повинен бути оголешений в області видимості не більше одного разу, чи те, що інструкція break розміщена в охоплюючому циклі інструкції switch, являються синтаксичними, хоча ці вимоги не кодуються і не забезпечуються граматикою, що вжита для користування.
- Перевірки типів. Правила типів мови гарантують, що оператор або функція буде застосована до коректного числа опреаторів допустимого типу. При необхідності приведення типів, наприклад при складанні цілого числа з числом із плаваючою

комою, програма перевірки типів може вставити відповідний оператор в синтаксичне дерево.

2.3 Методи оптимізації програмного коду

2.3.1 Метод усунення спільних підвиразів

Одним з різновидів оптимізацій для компіляторів є усунення загальних підвиразів (англ. Common subexpression elimination або CSE). Суть цього методу оптимізації полягає у пошуку в програмі обчислення, що застосовується більше одного разу на поточній ділянці та видаленні другої і наступної однакової операції, якщо це є можливим та раціональним з точки зору затрат ресурсів. Даний вид оптимізації вимагає проведення аналізу потоку даних для знаходження надлишкових обчислень і практично завжди покращує час виконання програми при застосуванні.

Підвираз в програмі має називається загальним, якщо існує інший такий же підвираз, який завжди обчислюється перед обчисленням даних, і операнди якого не змінюються в проміжку між обчисленнями. Наприклад, розглянемо загальний підвираз b * c.

```
a = b * c + g;
d = b * c * d;
```

Виходячи з даного визначення, усунення загальних підвиразів - це перетворення, яке знищує повторні обчислення загальних підвиразів і замінює їх на використання значення, що було збережене після першого обчислення. Однак, в розглянутому прикладі можна відразу при обчисленні d замінити загальний підвираз на значення змінної a, тому що дана змінна може змінюватися між розглянутими обчисленнями.

До прикладу загального підвиразу b * c застосуємо перетворення, що матимуть вигляд:

```
tmp = b * c;
a = tmp + g;
```

d = tmp * d;

Перетворення будуть ефективними, якщо загальний час запису і декількох читань нової змінної "tmp" виявиться меншим, ніж загальний час, що витрачається на обчислення виразу "b * c" кожен раз, коли він зустрічається в коді.

Розрізняють два види даної оптимізації:

- локальне усунення загальних підвиразів, яке працює в межах однієї лінійної ділянки
- глобальне усунення загальних підвиразів, яке працює в межах цілої процедури

Обидва види даної оптимізації засновані на аналізі потоку даних в ході якого визначається доступність виразів в кожній точці програми.

Саме на аналізі доступних виразів базується застосування оптимізації. Вираз x + y є доступним в деякій точці коду програми p, якщо: уздовж будь-якого шляху від початкового вузла до p вираз x + y обчислюється до досягнення цієї точки p; між обчисленнями виразів і досягненням точки p не відбувається зміни значень змінних x і y.

Ефективність перетворення головним чином залежить від того, що загальний час запису і декількох читань нової змінної "tmp" виявляється меншим за загальний час, що витрачається на обчислення виразу "b * c" кожен раз, коли воно зустрічається в коді. На практиці на підсумкову ефективність впливає також ряд інших факторів, зокрема розподіл змінних по регістрах. Розглянемо алгоритм оптимізації усуненням загальних підвиразів:

1. Для перетворень коду потрібно перевести його синтаксичне дерево, або в ще більш зручне представлення коду, наприклад семантичне дерево.
2. Суми потрібно перевести з бінарних в n-арні операції. Якщо одна з складових - сума, потрібно прибрати цю складову і

додати її підскладову. Встановлюється канонічний порядок на доданках, щоб не розрізняти x + 3 і 3 + x.

3. Складається список всіх підвиразів в дереві шляхом його рекурсивного проходу.
4. Надалі потрібно ввести порядок на виразах. Наприклад, можна написати хеш-функцію, щоб однакові підвирази обов'язково отримували однакове значення.
5. Обчислюється значення хеша на кожному переході рекурсивно. Сортується список за хешу.
6. Розглядаються відносно невеликі групи з однаковим хешем. Визначається рекурсивно рівність виразів:
    a. висоти дерев не рівні =>підвирази нерівні
    b. типи коренів не збігаються =>підвирази нерівні
    c. піддерева не збігаються =>підвирази нерівні
    d. інакше рівні
7. Отримано список однаких підвиразів. Оскільки суми можна розділити на частини по-різному, обчислюються хеші також для часткових сум. Досить впорядкувати складові, і обчислити 2^n варіантів, включаючи та не включаючи кожен з доданків. Ці часткові суми теж додаються в список переходів з посиланням на оригінальний вираз.

Для простих випадків, таких як було розглянуто вище, усувається дублювання обчислень арифметичних виразів. Найбільше значення даної оптимізації має її застосування для внутрішнього представленя компілятора, наприклад, при обчисленні індексів масиву, де ручна оптимізація виявляється сильно ускладненою або неможливою. У деяких мовах програмування можливе створення безлічі однакових обчислень. Наприклад, макроси мови С, які у вихідному коді не утворюють однакових виразів, проте після заміни імені макросу при обробці препроцесором на

послідовність програмних інструкцій, можлива поява такої безлічі обчислень.

У разі глобального застосування оптимізації на покращення впливають додаткові критерії. Наприклад, варто враховувати лічильники виконання базових блоків, так як, зменшивши статичну кількість обчислень виразу, можна ненароком збільшити динамічну, що впливає негативно на кількість виконаних операцій в програмі. Це призводить до того, що може бути вигідно використовувати зворотну оптимізацію.

2.3.2 SSA

SSA (англ. Static single assignment form) – називають проміжне представлення, що використовується компіляторами, де для кожної змінної присвоюється значення лише один раз. Змінні вихідної програми розбиваються на версії, зазвичай за допомогою додавання суфікса, таким чином, що кожне присвоювання здійснюється оригінальною версією змінної. У SSA-переставленні DU-ланцюги (англ. Def-use) задані явно і містять єдиний елемент.

Перевагою використання проміжного представлення є те, що для коду, записаного в SSA-формі простіше і ефективніше проводити багато видів компіляційної оптимізації. Наприклад, в наступному коді:

```
y := 1
y := 2
x := y
```

стає очевидним, що перше присвоювання не потрібно, так як значення y, використане в третьому рядку, відповідає другому присвоюванню. Однак для того, щоб з'ясувати це, компілятору довелося б вдатися до аналізу результуючих визначень. Але з використанням SSA-представлення це стає набагато простіше:

```
y1 := 1
y2 := 2
x1 := y2
```

SSA робить можливими або істотно спрощує такі оптимізаційні алгоритми:

- згортка констант
- видалення «мертвого» коду
- нумерація глобальних значень
- часткове усунення надмірності
- зниження вартості операцій
- розподіл регістрів

У завдання аналізу потоку управління входить визначення властивостей передачі управління між операторами програми. Перевірка багатьох властивостей цього виду необхідна для вирішення завдань оптимізації, перетворень програм та ін. При вирішенні цих завдань зазвичай використовується формальна графова модель (тобто аналіз проводиться над графом потоку управління), а самі завдання формулюються в теоретико-графовой формі.

Основне вживання аналізу потоку керування в оптимізації - це фрагментація, тобто уявлення графа потоку керування у вигляді сукупності фрагментів певного виду. Таке уявлення необхідно для застосування практично всіх оптимізують перетворень.

Основним способом подання потоку управління програми є граф потоку керування - орієнтований граф з двома виділеними вершинами start і stop, такими, що:

- в start не входить жодна дуга
- з stop не виходить жодна дуга
- довільна вершина належить хоча б одному шляху з start в stop

Переклад звичайного програмного коду в SSA-уявлення досягається шляхом заміни в кожної операції присвоювання змінної з лівої частини на нову змінну. Для кожного використання значення змінної вихідне ім'я

замінюється на ім'я «версії», відповідної потрібного базового блоку. Розглянемо наступний приклад графу потоку керування на рисунку 2.6.

Відповідно до визначення SSA створимо замість змінної x дві нові змінні x1 і x2. Кожній з них значення буде присвоєно рівно один раз. Аналогічним чином замінимо інші змінні, після чого отримаємо наступний граф, другий приклад показаний на рисунку 2.7.

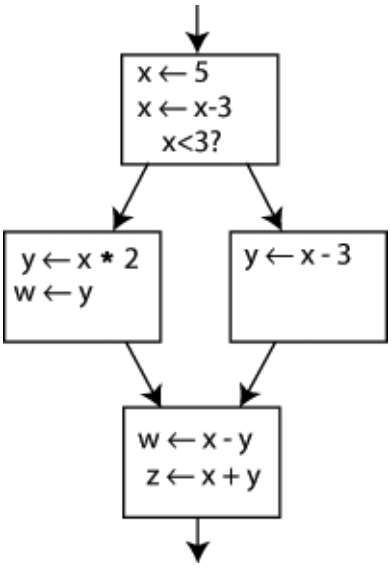


Рис. 2.6 - Перетворення коду в SSA-уявлення. Приклад 1

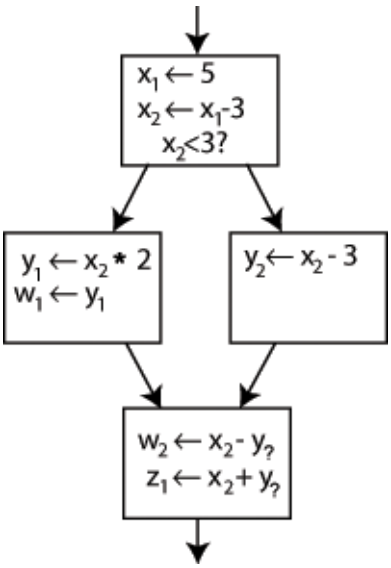


Рис. 2.7 - Перетворення коду в SSA-уявлення. Приклад 2

І поки залишається неясним, яке значення y буде використовуватися в нижньому блоці. Там ім'я y може означати як y1, так і y2. Для того, щоб дозволити неоднозначность такого роду, в SSA введена спеціальна Ф-функція. Ця функція створює нову версію змінної y, якій буде присвоєно

значення або з y1, або з y2, в залежності від того, з якої гілки перейшло управління, що зображено на рисунку 2.8 на прикладі 3.

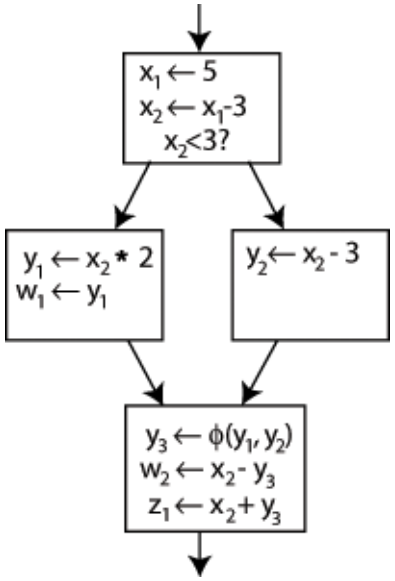


Рис. 2.8 Переклад коду в SSA-представлення. Приклад 3

При цьому використовувати Ф-функцію для змінної x не потрібно, так як лише одна версія x (а саме, x2) «досягає» останнього блоку.

Ф-функція в дійсності не реалізована; вона являє собою лише вказівку компілятору зберігати всі змінні, перераховані в списку її аргументів, в одному і тому ж місці в пам'яті (або регістрі).

2.3.3 Згортка констант

Іншим видом оптимізацій для компіляторів є метод, заснований на принципах згортки констант. Згортка констант - добре відома проблема глобального аналізу потоку даних. Мета поширення констант полягає в виявленні величин, які є постійними при будь-якій можливій ситуації виконання програми, і в поширенні цих величин так далеко по тексту програми, як тільки це можливо. Вирази, чиї операнди є константами, можуть бути обчислені на етапі компіляції. Тому використання алгоритмів поширення констант дозволяє компілятору видавати більш раціональний і оптимізований код.

Хоча в загальному випадку проблема поширення констант є нерозв'язною, існує безліч проявів цієї проблеми, для яких існують ефективні алгоритми.

Технології згортки констант дозволяють вирішити такі задачі:

- вирази, що обчислюються на етапі компіляції не потрібно обчислювати в процесі виконання програми
- код, який ніколи не виконується, може бути видалений. Недосяжний код може з'являтися в тих випадках, коли значення предиката в умовному вираженні незмінно і відомо на етапі компіляції
- впровадження процедур

Згортка констант спільно зі впровадженням процедур (коли багато параметрів процедур виражені константами) дозволяє уникнути розростання коду, яке часто є результатом впроваджених процедур в місця з викликів.

Алгоритм поширення констант використовує SSA-представлення програми. У SSA-форму програма трансформується таким чином, що тільки одне присвоювання може досягати точки виконання.

2.3.4 Видалення мертвого коду

Метод видалення мертвого коду є досить вагомою оптимізацією, проте навіть дуже незначні його реалізації можуть призводити до непередбачуваної поведінки, що потягне за собою велику кількість помилок.

В теорії компіляторів видаленням мертвого коду називається оптимізація, що видаляє непотрібний код. Мертвим кодом (так само марним кодом) називають код, виконання якого не впливає на висновок результату програми, всі результати обчислення такого коду є мертвими змінними, тобто змінними, значення яких в подальшому в програмі не використовуються.

У зв'язку з існуючим тлумаченням терміну «мертвий код», важливо відзначити, що оптимізація видалення мертвого коду не займається видаленням недосяжного коду. Локалізацією і видаленням недосяжного коду можуть займатися збирачі програмного «сміття» або інші оптимізації, наприклад, видалення недосяжного коду.

Вилучення з програми непотрібного коду здатне пришвидшити час виконання програми за рахунок зменшення кількості виконуваних в ній операцій і зменшити розмір програми або проміжного представлення. Розглянемо приклад коду на мові програмування С на рисунку 2.9. В даному прикладі операція додавання var2 = var1 + 3 є мертвим кодом, так як змінна var2 не використовується в подальших обчисленнях і є мертвою, починаючи з цієї точки і закінчуючи кінцем процедури. Після видалення цієї інструкції отримаємо мертву змінну (рисунок 2.10).

```
1  int func(void)
2  {
3      int var1 = 24;
4      int var2;
5      var2  = var1 + 3; /*Мертвий код */
6      return var1 << 2;
7  }
```

Рис. 2.9 - Приклад мертвого коду

```
1  int func(void)
2  {
3      int var1 = 24;
4      int var2; /*Мертва змінна*/
5      return var1 << 2;
6  }
```

Рис. 2.10 Приклад мертвої змінної

Незважаючи на те, що обчислення відбувається всередині функції, його результат записується в змінну, що знаходиться в області видимості тільки цієї функції; і якщо врахувати що функція безумовно повертає число 96, вона може бути спрощена оптимізацією поширення констант так,

щоб її тіло складалося тільки з операції return 96. А потім компілятор може замінити всі виклики цієї функції на значення, що повертається.

Класичний алгоритм DCE працює на SSA-представленні і складається з двох проходів: перший прохід відзначає свідомо «живі» операції (операції виходу з процедури, введення-виведення, що змінюють глобальні змінні); другий обхід починається з живих операцій і йде вгору за визначеннями аргументів, позначаючи всі операції на своєму шляху «живими», закінчуючи тими операціями, які не мають попередників в SSA-формі.

DCE може не проводити ніякого самостійного аналізу, а просто скористатися результатами аналізу активних змінних, але обчислювальна складність такого аналізу складає n^3 в гіршому випадку. Існують специфічні алгоритми, що займаються видаленням порожніх вузлів в графі, об'єднанням декількох базових блоків в один та інше, для такого аналізу потрібен побудований граф потоку керування.

Видалення мертвого коду може застосовуватися кілька разів в процесі компіляції, так як мертвий код знаходиться в програмі не тільки через те, що він міститься у вихідному коді - деякі інші перетворення здатні робити код мертвим. Наприклад, оптимізації розповсюдження копій і поширення констант часто перетворюють інструкції на безкорисні. Також доводиться видаляти мертвий код, створений іншими перетвореннями в компіляторі. Наприклад, класичний алгоритм оптимізації заміщає трудоємкі операції менш трудоємкими, після чого видалення мертвого коду усуває старі операції і завершує перетворення.

2.4 Програмний паралелізм обробки даних

Можна стверджувати, що існує досить багато різних технологій паралельного виконання програмного коду. Причому ці технології відрізняються не стільки мовами програмування, а скільки архітектурними підходами до побудови паралельних систем. Наприклад, якісь технології припускають побудову паралельних рішень на основі декількох

комп'ютерів (як одного, так і різних типів), інші ж припускають саме роботу на одній машині з декількома процесорними ядрами.

Системи на базі декількох комп'ютерів відносять до класу систем для розподілених обчислень. Подібні рішення використовуються досить давно, їх добре розуміють професіонали індустрії, стосовно їх є досить багато літератури. Найбільш яскравий приклад технології розподілених обчислень - MPI (Message Passing Interface - інтерфейс передачі повідомлень). MPI є дуже росповсюдженим стандартом інтерфейсу обміну даними в паралельному програмуванні. MPI може похвалитися безліччю реалізацій на різних платформах. Цей інтерфейс надає програмісту єдиний механізм взаємодії частин всередині розпаралеленого програмного продукту незалежно від використаної архітектури при його написанні (однопроцесорні / багатопроцесорні із загальною / роздільною пам'яттю).

Недоліком MPI є те, що він призначається для систем з використанням розділеної пам'ятті, тому його використання для організації паралелізму в системі із загальною пам'яттю є недоцільним. Це може вилитися у складний процес, де рішенням стане OpenMP. Заборони робити MPI-рішення для однієї машини немає, проте системи паралельного програмування для роботи на одній машині почали розроблятися відносно недавно. Такі ідеї не є новими, але саме з приходом багатоядерних системрозробникам варто звернути свою увагу на такі технології як OpenMP, Intel Thread Building Blocks, Microsoft Parallel Extensions та інші.

Варто зазначити, що технологія паралельного програмування повинна підтримувати можливість робити програму розпаралеленою поступово. Дійсно, ідеальну паралельну програму варто одразу писати з використанням технологій паралелізму та на функціональній мові. Проте на практиці розробники замість впровадження нової функціональної мови F# обирають добре знайому C ++, а то і взагалі мову C, де написаний код потребує розпаралелення. В такому випадку технологія OpenMP буде дуже

вдалим вибором. Вона дозволяє раціонально обрати місця програми для розпаралелення, шукає вузькі місця до тих пір, поки не буде отримана бажана продуктивність програми. Стає легшою організація процесу розробки, наприклад випуски проміжних релізів. Саме тому технологія OpenMP стала досить популярною.

2.4.1 Технологія OpenMP

OpenMP (Open Multi-Processing) – технологія представлена набором директив компілятора, бібліотечних процедур і змінних, що використовуються для програмування багатопотокових програмних засобів на багатопроцесорних системах із загальною пам'яттю (SMP-системах) [19 - 20]. Перший стандарт OpenMP було розробленно в 1997 році як API, орієнтований на багатопотокові програми з легкою переносимістю. Спочатку він був заснований на мові Fortran, але пізніше з’явився на мовах C і C++ та інших. OpenMP як інтерфейс став популярною технологією паралельного програмування. OpenMP успішно використовується як при програмуванні суперкомп'ютерних систем з великою кількістю процесорів, так і в звичайних персональних комп’ютерах. Розробку специфікації OpenMP ведуть кілька великих виробників обчислювальної техніки і програмного забезпечення.

У OpenMP використовується модель паралельного виконання "злиття гілок". Спочатку програма OpenMP починає своє виконання як єдиний потік, який називається початковим потоком. Коли потік зустрічається з паралельною конструкцією, відбувається створення нової групи потоків, що складається з деякого числа додаткових потоків, поточний потік займає посаду головного у новій групі. Завдання усіх членів нової групи (включаючи головну) полягає у виконанні коду всередині паралельної конструкції. В кінці паралельної конструкції присутній неявний бар'єр. Після паралельного виконання конструкції і призначеного для користувача коду, продовжує виконання тільки головний потік. В паралельній області можуть вкладатися інші паралельні області, в

яких кожен потік початкової області стає основним для своєї групи потоків. Вкладені області можуть в свою чергу включати області більш глибокого рівня вкладеності.

2.4.2 Технологія CUDA

Технологія CUDA - це представлена програмно-апаратна обчислювальна архітектура, розроблена компанією NVIDIA, що є заснованою на розширенній версії мови C [21]. Дана архітектура надає можливість організації доступу до набору інструкцій графічного прискорювача та можливістю управління його пам'яттю при виконанні паралельних обчислень. Варто зазначити, що складність процесу програмування коду для виконання на GPU за допомогою CUDA досить висока, проте вона все одно нижче, ніж попередня версія GPGPU рішення. Такі програми вимагають розбиття програмного комплексу між декількома мікропроцессорами, подібно до інтерфейсу MPI програмування, але без застосування поділу даних, які розташовані в загальній відеопам'яті.

Так як технологія CUDA застосовується для кожного з ядер мультипроцессора подібно до OpenMP, вона вимагає безпосереднього розуміння організації пам'яті. Хоча складність перенесення та розробки на CUDA має пряму залежність від складності програми.

Засіб CUDA має безліч зразків коду і добре задокументований для програмістів.

До основних характеристик CUDA відносять:

- уніфіковане програмно-апаратне рішення для паралельних обчислень на відеочіпах NVIDIA
- значний набір готових рішень, від мобільних до мультичіпових пристроїв
- підтримується мова програмування C
- наявні стандартні бібліотеки чисельного аналізу FFT (швидке перетворення Фур'є) і BLAS (лінійна алгебра)

- оптимізований обмін даними між CPU і GPU
- взаємодія з графічними API OpenGL і DirectX
- підтримка 32- і 64-бітових операційних систем
- виконання програмного коду на низькому рівні

До основних переваг CUDA при порівнянні з попередніми методами GPGPU можна віднести проектування архітектури, що дає ефективне використання обчислень на GPU і використання мови програмування C, за допомогою якої не вимагається перенесення алгоритмів в зручний для графічного прискорювача вид.

Також CUDA надає апаратні можливості, що є недоступними з графічних API, такі як колективна пам'ять. Сюди відноситься пам'ять невеликого обсягу, що доступна блокам потоків. Вона дозволяє кешувати найбільш часто використовувані дані і може надавати більш високу швидкість, в порівнянні з використанням методу текстурних вибірок для цього завдання. Це в свою чергу, знижує чутливість до пропускної спроможності паралельних алгоритмів у багатьох програмах. Наприклад, відчутний ефект прискорення виконання коду у реалізаціях алгоритмів лінійної алгебри, швидкого перетворення Фур'є і фільтрів обробки зображень.

Зручний в CUDA і доступ до самої пам'яті. Програмний код в графічному API виводить дані у вигляді 32-розрядних значень з плаваючою комою в заздалегідь заготовлені області, а CUDA підтримує необмежене число записів за будь-якою адресою. Такі переваги роблять можливим виконання на GPU деяких алгоритмів, які неможливо ефективно реалізувати за допомогою методів GPGPU, заснованих на графічних API.

Також, графічні API в обов'язковому порядку зберігають дані в структурах, що вимагає попередньої підготовки великих масивів для їх зберігання, що ускладнює алгоритм і змушує використовувати спеціальну

адресацію, а CUDA дозволяє читати дані за будь-якою адресою. Ще однією перевагою CUDA є оптимізація в обміні даними між CPU і GPU. А для розробників, бажаючих отримати доступ до низького рівня CUDA пропонує можливість низькорівневого програмування на асемблері.

2.5 Висновок до другого розділу

Перевага крос-платформленості за рахунок використання байт-коду, виконання якого під керівництвом віртуальної машини значно повільніша за традиційне виконання машинного коду, нівелюється у алгоритмах з інтенсивними обчисленнями.

У даному розділі розглянуто застосування динамічної компіляції для поєднання найкращих особливостей двох підходів виконання програмного коду, компіляції та інтерпретації. За допомогою динамічної компіляції стає можливим переносити виконання певних ділянок коду з інтепретатора напряму до центрального процесора за рахунок перетворення таких ділянок у машинний код. Таким чином, вдається зменшити розрив у швидкодії виконання програмного коду інтерпретатором у порівнянні з компіляцією.

Також розглянуто види оптимізації компіляції, за рахунок яких вдається досягнути значного покращення показників продуктивності та швидкості виконання програмного коду.

Допоміжним і важливим інструментом із розпаралеленням програм виступає технологія OpenMP, що дає унікальну можливість програмувати багатопотокові програми на багатопроцесорних системах. У першому розділі описано переваги та принципи роботи технології OpenMP.

Варто зазначити, що виконання програм здійснюється швидше на графічних процесорах, ніж на центральних, тому програмно-апаратна обчислювальна архітектура NVIDIA – CUDA є досить потужною технологією, що дозволяє задіяти у роботі саме графічні прискорювачі.

3 ПРИНЦИПИ ТА ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ РОЗРОБЛЕНОГО МЕТОДУ ДИНАМІЧНОЇ КОМПІЛЯЦІЇ PYTHON ПРОГРАМ, ОРІЄНТОВАНИХ НА ОБРОБКУ МАСИВІВ

Розроблений метод динамічної компіляції є надбудовою над стандартним інтерпретатором мови Python, він вибірково перехвачує виконання відмічених користувачем функцій. Якщо користувач хоче скомпілювати якусь функцію, йому необхідно позначити дану функцію декоратором @jit, наприклад, розглянемо рисунок 3.1:

```
1  @jit
2  def avg(X,Y):
3      return (X+Y) / 2.0
4
```

Рис. 3.1 - Знаходження середнього значення з двох масивів

Якщо прибрати декоратор @jit з вище вказаного прикладу, тоді функція avg виконалася би як звичайний Python код. Без застосування динамічної компіляції функція avg була би трансльована у байт-код Python інтерпретатором з подальшим використанням методів обробки масивів __add__ та __divide__. Так як у NumPy оператори додавання та розподілу масивів скомпільовані окремо, обов'язково відбувається виділення ще одного масиву для зберігання проміжних результатів. У даному випадку виділення такого масиву є марною втратою ресурсів так як результат додавання двох масивів одразу ж використовується для їх розподілу.

Розроблений метод спеціалізує функцію avg для переданого до неї набору вхідних даних, виконує оптимізацію тіла функції у єдину скомбіновану операцію над масивами для уникнення непотрібних ресурсозатратних проміжних виділень та перетворює вихідний програмний код у низькорівневий машиний код, який виконується у паралельних потоках.

Декоратор @jit не може бути застосований для перехвату виконання будь-якої Python функції, так як розроблений метод не є компілятором мови Python загального застосування. Задача декоратору полягає у перехваті викликів позначеної функції та ініціації наступного переліку подій:

1. Транслювання функції у не типізоване представлення, яке буде змінюватися в залежності від типів вхідних параметрів функції для подальшого створення типізованої версії функції
2. Спеціалізація не типізованої функції для переданих до функції аргументів, створення типізованої версії функції
3. Виконання ряду оптимізацій вихідного програмного коду
4. Перетворення коду у машинний для виконання у паралельних потоках на графічному прискорювачі за допомогою технології CUDA або з використанням можливостей багатоядерних центральних процесорів за допомогою технології OpenMP.

Розроблений метод підтримує певний набір типів даних Python, а саме: числа, словники, списки та масиви NumPy. Для роботи з цими типами даних розробник може використовувати будь-яку стандартну математичну або логічну операцію, а також частину вбудованих у бібліотеку NumPy функцій.

На відміну від звичайних засобів мови Python, кожна змінна величина у проміжному представленні визначається з вказанням її типу. Крім того, такі оператори як наприклад додавання є значно більш статичними, ніж аналоги мови Python. Наприклад, на відміну від мови Python, у якій оператор додавання виконує операції над динамічним представленням методу __add__, у запропонованому методі кожен виклик оператору додавання виконується на числових значеннях певного визначеного типу. Арифметичні операції над елементами масивів мають бути використані з застосуванням операторів для виконання коду у

паралельних потоках, прикладом такого оператору є **Map.** Наприклад, якщо функція з рисунку 3.1 була б викликана з переданими до неї аргументами з типом векторів з плаваючою комою, тоді проміжне представлення такої функції виглядало би наступним чином (рисунок 3.2):

```
1  def avg(X :: array1(float32), Y :: array1(float32)):
2      temp :: float32 = Map(lambda xi, yi: xi + yi, X, Y)
3      return Map(lambda xi: xi / 2.0, temp)
4
```

Рис. 3.2 - Проміжне типізоване представлення функції знаходження середнього арифметичного значення

Паралельне виконання коду досягається за рахунок використання наступних операторів та їх різновидів:

- **map** (функція, масив аргументів, вісь) – виконання функції для кожного елементу масиву аргументів. По замовчанню значення вісі дорівнює None, що означає що функція f виконується на всіх числових елементах масиву аргументів. Якщо у якості аргументу вісі буде вказано цілочисельне значення, тоді у якості аргументів до вказаної функції будуть передані зрізи відповідної розмірності. Це може бути використано для застосування функції до всіх рядків або стовпців матриці.
- **reduce** (функція, масив аргументів, початкове значення, вісь) – комбінування усіх елементів масиву аргументів використовуючи рекурсивну передану функцію. Є можливість вказати початкове значення для переданої рекурсивної функції. Прикладами даного оператору є реалізація NumPy функцій sum та product, коли застосовується даний оператор для додавання або множення всіх елементів переданого масиву, запускається

рекурсивна функція яка відповідно рекурсивно додаває або множить усі елементи (комбінує їх між собою).

- **scan** (функція, масив аргументів, початкове значення, вісь) – відбувається комбінування всіх елементів масиву вхідних аргументів та повертається масив, у якому знаходяться усі проміжні дані обчислень від застосування рекурсивної функції. Є можливість вказати початкове значення для переданої рекурсивної функції. По замовчанню значення вісі дорівнює None, що означає що функція f виконується на всіх числових елементах масиву аргументів. Якщо у якості аргументу вісі буде вказано цілочисельне значення, тоді у якості аргументів до вказаної функції будуть передані зрізи відповідної розмірності. Це може бути використано для застосування функції до всіх рядків або стовпців матриці.

Для кожної появи такого оператору у вихідному коді програми синтезується код, що виконується у паралельних потоках, котрий реалізує призначення використаного оператору відповідно до аргументів переданих до нього.

У розробленому методі використовується семантика розширення розмірів масивів представлена у бібліотеці NumPy, це означає що при арифметичних операціях над масивами різних розмірів, менший з них "розширюється" до розмірів більшого, ця операція виконується без створення зайвих копій даних та зазвичай приводить до більш ефективного виконання алгоритмів за рахунок того, що таким чином для арифметичних операцій використовується менше операцій з пам'яттю. Бібліотечні функції NumPy перероблені для використання описаних операторів паралельного виконання коду.

3.1 Етапи перетворення програмного коду

Розглянемо процес трансформації та виконання програмного коду на наступному прикладі (рисунок 3.3):

```
1  @jit
2  def norm(x):
3      return np.sqrt(sum(x*x))
4
```

Рис. 3.3 - Нормалізація вектору

Коли інтерпретатор Python доходить до оголошення функції norm, декоратор ініціює процес парсингу вихідного коду функції та транслює його у не типізоване проміжкове представлення. Певний набір вбудованих функцій, таких як sum, та функцій NumPy, як наприклад np.sqrt, транслюється безпосередньо у власне синтаксичне представлення.

Якщо декоратором буде позначена функція, яка не може бути трансльована або суперечить семантичним обмеженням розробленого методу, то при виконанні такого коду буде з'являтися помилка виконання.

Коли функція norm викликається відбувається перехват виклику до неї та створюється типізована версія даної функції. При процесі типізації всі функції, які викликаються функцією norm, також проходять процес визначення типів для кожного окремого набору типів вхідних аргументів. В нашому випадку відповідно до прикладу на рисунку 3.3, якщо функція norm була б викликана з переданим до неї одномірним масивом з типом елементів з плаваючою комою, тоді тип sum також був б визначеним для такого ж самого типу вхідних даних, тоді як анонімна функція створення для додавання елементів x*x була б типізована для пар чисел з плаваючою комою.

Для типізованого представлення кожна функція має мати чіткі однозначні типи для вхідних та вихідних даних. Для позбавлення поліморфізму відбувається конвертація типів та використання оператору

Map там де це необхідно. Коли функція norm типізована для вхідного векторного типу, використання оператору множення перетворюється у множення одномірного масиву чисел за допомогою оператору Map.

Код проходить декілька етапів оптимізації, які допомагають уникнути надлишкових обчислень, а саме:

- видалення спільних виразів – відбувається пошук обчислень, які виконуються більше одного разу на ділянці коду, що розглядається, такі обчислення видаляються та їх результати замінюються збереженними результатами першого обчислення
- вбудування функцій – відбувається пошук функцій, які можна поєднати між собою і виконати за допомогою лише одного виклику
- згортка констант – відбувається обчислення константних виразів на етапі компіляції, перш за все спрощуються констатні вирази, які містять числові літерали

Окрім стандартних оптимізацій компіляції, таких як видалення спільних виразів, вбудування функцій та згортка констант, також застосовується об'єднання операторів масиву, за рахунок цього підвищується обчислювальна інтенсивність коду та вдається запобігти виділенню непотрібних проміжкових значень.

Розроблений метод динамічної компіляції Python програм орієнтований на математично інтенсивні обчислення над масивами NumPy. Обсяг мовних конструкцій, що підтримується, є досить обмеженим у порівнянні зі звичайним Python кодом. Метод дозволяє ефективно трансформувати вихідний код програми у низькорівневий машиний, головною сферою призначення є такі наукові області, як: машине навчання, фінансові та економічні обчислення та наукові симуляції — у всіх цих напрямках широко поширені математичні операції над масивами

даних. Код, який може бути оброблений, має бути у рамках наступних правил:

- підтримуються наступні типи даних: кортежі, числа, зрізи, списки, масиви NumPy та об'єкт None. Інші типи даних, як наприклад словники, множини або об'єкти, створені користувачем, не підтримуються
- не підтримуються оператори контролю виконання програмного коду break та continue, також не підтримується використання винятків
- має бути доступ до вихідного коду функції, тобто попередньо скомпільовані розширення на мові C або функції, які викликають такі розширення, не можуть бути використані
- для того, щоб скомпілювати Python код у машиний, необхідно визначити тип кожного виразу. Для кожного набору типів вхідних аргументів створюється нова типізована версія функції, за рахунок цього зберігається певна частина поліморфізму. Однак вирази, типи яких залежать від динамічних величин не підтримуються

Далі буде детально розглянуто процес трансляції вихідного програмного коду простої функції у машиний код (рисунок 3.4).

```
1  def count(values, thresh):
2      n = 0
3      for i in range(0, len(values), 1):
4          (header)
5          n_loop <- phi(n, n2)
6          (body)
7          elt = values[i]
8          n2 = n_loop + (elt < thresh)
9      return n_loop
10
```

Рисунок - 3.4 Програмний код функції count

Функція, яка буде розглянута у процесі компіляції приведена на рисунку 3.4, дана функція count сумує кількість елементів масиву, які меньші за передане до цієї функції значення thresh. Перш ніж перейти безпосередньо до розбору компіляції функції count, розглянемо як вона виконується стандартним CPython інтерпретатором.

Перше за все інтерпретатор обробляє вихідний код функції. Цей код зчитується як строка символів, над цією строкою відбувається синтаксичний аналіз і повертається структуроване синтаксичне дерево представлене на рисунку 3.5.

```
FunctionDef(
    name='count',
    args=arguments(args=[Name(id='values'), Name(id='thresh')],
                vararg=None, kwarg=None, defaults=[]),
body=[
    Assign(targets=[Name(id='n')], value=Num(n=0)),
    For(target=Name(id='elt'), iter=Name(id='values'), body=[
        AugAssign(target=Name(id='n'), op=Add(),
            value=Compare(left=Name(id='elt'), ops=[Lt()],
                          comparators=[Name(id='thresh')]))]),
    Return(value=Name(id='n', ctx=Load()))])

```

Рис. 3.5 - Синтаксичне дерево функції count

Далі інтепретатор виконує команди відповідно до синтаксичного дерева. У мові Python досягли незначного прискорення швидкодії за рахунок компіляції у більш компактний байт-код показаний на рисунку 3.6.

```
 1   0   LOAD_CONST      1 (0)
 2   3   STORE_FAST      2 (n)
 3
 4   6   SETUP_LOOP      30 (to 39)
 5   9   LOAD_FAST       0 (values)
 6   12  GET_ITER
 7   13  FOR_ITER        22 (to 38)
 8   16  STORE_FAST      3 (elt)
 9
10   19  LOAD_FAST       2 (n)
11   22  LOAD_FAST       3 (elt)
12   25  LOAD_FAST       1 (thresh)
13   28  COMPARE_OP      0 (<)
14   31  INPLACE_ADD
15   32  STORE_FAST      2 (n)
16   35  JUMP_ABSOLUTE   13
17   38  POP_BLOCK
18
19   39  LOAD_FAST       2 (n)
20   42  RETURN_VALUE
21
```

Рис. 3.6 - Python байт-код функції count

Низька ефективність інтерпретаторів, які виконують команди згідно синтаксичного дерева, є однією з причин чому мова Ruby в загалом буда повільнішою за Python. Не зважаючи на те, що використання байт-коду є більш ефективним у плані швидкодії у порівнянні з проходженням по синтаксичному дереву, виконання байт-коду все одно дає дуже погані результати швидкодії виконання програми.

Отже є два варіанти отримання представлення функції, яке може виконуватися, байт-код та синтаксичне дерево. Є вагомі причини для того, щоб не використовувати синтаксичне дерево, наприклад той факт, що синтаксичне дерево ніде не зберігається і кожного разу має бути відтворене з вихідного коду. Однак байт-код використовує багато непотрібних операцій зі стеком та не зберігає деякі конструкції високого рівня. Отже, хоча синтаксичне дерево і є кращим вибором у даному

випадку, воно всеодно є незручнім для аналізу та трансформації програми, воно береться за основу та перетворюється у подальше не типізоване представлення. Розглянемо рисунок 3.7.

```
1  def count(values, thresh):
2      n = 0
3      for i in range(0, len(values), 1):
4          (header)
5              n_loop <- phi(n, n2)
6          (body)
7              elt = values[i]
8              n2 = n_loop + (elt < thresh)
9      return n_loop
10
```

Рис. 3.7 Проміжне нетипізоване представлення функції count

На рисунку 3.7 показано як виглядає проміжне нетипізоване представлення функції count. Лічильник циклу n був розділений на 3 окремі частини: n, n2 та n_loop, таким чином програма була трансльована у SSA представлення. SSA (англ. Static single assignment form) - проміжне представлення, яке використовується компіляторами, в якому для кожної змінної значення присвоюється лише один раз. Змінні вихідної програми розбиваються на версії, зазвичай за допомогою додавання суфікса, таким чином, що кожне присвоювання здійснюється на унікальній версії змінної. Даний підхід схожий на функціональне програмування. Для коду в SSA-формі простіше і ефективніше проводити багато видів компіляторної оптимізації.

Коли відбувається виклик нетипізованої функції, вона клонується для кожного окремого набору вхідних типів даних. Типи інших змінних, які не передаються у вигляді параметрів, а знаходяться у тілі функції, конвертуються за необхідності. Розглянемо рисунок 3.8.

```
def count(values :: array1(float64), thresh :: float64) =>
    int64:
        n :: int64 = 0 :: int64
        shape_tuple :: tuple(int64) = values.shape
        for i in range(0, shape_tuple[0], 1):
            (header)
                n_loop <- phi(0 :: int64, n2)
            (body)
                elt :: float64 = values[i]
                less_tmp :: bool = elt < thresh
                n2 :: int64 = n_loop + cast<int64>(less_tmp)
    return n_loop

```

Рис. 3.8 - Типізоване проміжкове представлення функції count

Типізована версія функції count показана на рисунку 3.8. Типи вхідних даних функції були спеціалізовані як array1(float64) та float64, а тип значення, яке повертається функцією, визначений як int64. Булеве проміжне значення, яке виникає при порівнянні чергового елементу масиву конвертується у int64 перед тим як додаватися до змінної n2. Якщо використовувати одну змінну у одних випадках як масив, а у інших як число, виникне помилка виконання.

Визначення типів вже надає суттєве прискорення швидкодії виконання програмного коду за рахунок того, що надає розгорнуте представлення чисел. Додавання двух чисел з плаваючою комою, які знаходяться у відповідних їм регістрах відбувається значно швидше, ніж Python виклик операції __add__ для двух об'єктів PyFloatObjects.

Однак, навіть після етапу визначення типів даних у коді, його виконання всеодно було би значно більш повільним ніж аналогічний варіант на мові програмування низького рівня. Також відбувається набір стандартних оптимізацій компіляції, таких яквидалення спільних виразів, вбудування функцій та згортка констант. Окрім цього, для зниження витрат обчислювальних ресурсів на подібних виразах 0.5*array1 + 0.5*array2 відбувається об'єднання операцій над масивами, що надає ще більший потенціал для оптимізації виконання програмного коду.

```
def count(values :: array1(float64), thresh :: float64) =>
 int64:
    shape_tuple :: struct(int64) = values.shape
    data :: ptr(float64) = values.data
    base_offset :: int64 = values.offset
    for i in range(0, shape_tuple.elt0, 1):
        (header)
            n_loop <- phi(0 :: int64, n2)
        (body)
            offset :: int64 = offset + i
            elt :: float64 = data[offset]
            less_tmp :: bool = elt < thresh
            n2 :: int64 = n_loop + cast<int64>(less_tmp)
    return n_loop

```

Рис. 3.9 - Оптимізоване проміжне представлення функції count

У даному випадку, розглянутому на рисунку 3.9, обчислення є досить простим, отже є всього декілька можливостей оптимізації. Окрім переписання коду для більш швидкого його виконання, такі конструкції як масиви або кортежі перетворюються у більш примітивні. У розглянутому лістингу коду не відбувається прямих звернень по індексу n-вимірного масиву, а окремо обчислюється зміщення індексу та потім воно передається до вказівника даних цього масива. Даний процес перетворення складних конструкцій спрощує наступний етап перетворення програми: транслювання у код мови С.

Трансляція з отриманого оптимізованого проміжного представлення функції з пониженням конструкцій високого рівня у мову С є в основному механічною. Структуровані типи даних, такі як дескриптори масивів та кортежі, перетворюються на тип структур мови С і за рахунок того, що дані у них не змінюються, вони можуть бути передані по своїм значенням. Для того, щоб цей код можна було використати інтерпретатором мови Python, у момент входження у скомпільовану С функцію необхідно дістати всі вхідні величини з представлення PyObject у інтепретаторі Python. Також відповідно результат має бути повернений таким чином, щоб інтепретатор Python його зрозумів (рисунок 3.10).

Програмний код функції count згенерований для мови програмування C показаний на рисунку 3.11. В даному випадку вибір бекенду для виконання цього коду майже не впливає на швидкість виконання, так як відсутні оператори паралельного виконання коду у початковому коді функції, отже в незалежності від того, де код буде виконуватися, отримані результати швидкості виконання будуть майже однаковими.

Коли код передається на компілятор мови C, робота розробленого методу є майже повністю виконаною. Зовнішній компілятор виконує свій ряд оптимізацій над переданим до нього кодом, та нарешті на виході отримується машинний код, його частина представлена на рисунку 3.10.

```
;; loop entry
    movq 8(%rdi), %rax
    movq (%rax), %r8
    xorl %eax, %eax
    testq %r8, %r8
    jle .LBB0_3
;; %loop_body.preheader
    movq 24(%rdi), %rax
    movq (%rdi), %rdx
    leaq (%rdx,%rax,8), %rdx
    xorl %eax, %eax
    .align 16, 0x90
;; %loop_body
.LBB0_2:
    ucomisd (%rdx), %xmm0
    seta %cl
    movzbl %cl, %esi
    addq %rsi, %rax
    addq $8, %rdx
    decq %r8
jne .LBB0_2
.LBB0_3:
    ret
```

Рис. 3.10 - Згенерований машинний код x86 основного циклу функції count

```c
#include <Python.h>
#include <numpy/arrayobject.h>
#include <numpy/arrayscalars.h>
#include <stdint.h>
#include <math.h>

typedef struct float64_ptr_type {
    double* raw_ptr;
    PyObject* base;
} float64_ptr_type;

typedef struct array_type {
    float64_ptr_type data;
    npy_intp shape[1];
    npy_intp strides[1];
    int64_t offset;
    int64_t size;
} array_type;

PyObject* count (PyObject* dummy, PyObject* args) {
    PyObject* values = PyTuple_GetItem(args, 0);
    npy_intp* shape_ptr = PyArray_DIMS((PyArrayObject*)values);
    npy_intp* strides_bytes = PyArray_STRIDES( (PyArrayObject*)
        values);
    array_type unboxed_array;
    float64_ptr_type data = {(double*) PyArray_DATA(((
        PyArrayObject*) values)), values};
    unboxed_array.data = data;
    unboxed_array.strides[0] = strides_bytes[0] / 8;
    unboxed_array.shape[0] = shape_ptr[0];
    unboxed_array.offset = 0;
    unboxed_array.size = PyArray_Size(values);
    PyObject* thresh = PyTuple_GetItem(args, 1);
    double thresh_2;
    if (PyFloat_Check(thresh)) { thresh_2 = PyFloat_AsDouble(
        thresh); }
    else { PyArray_ScalarAsCtype(thresh, &thresh_2); }
    npy_intp* shape = unboxed_array.shape;
    int64_t len_result = shape[0];
    int64_t n_loop = 0;
    int64_t idx;
    for (idx = 0; idx < len_result; idx += 1) {
        double elt = unboxed_array.data.raw_ptr[idx];
        int64_t temp = ((int64_t) (elt < thresh_2));
        int64_t n_loop = n_loop + temp;
    }
    return (PyObject*) PyArray_Scalar(&n_loop,
        PyArray_DescrFromType(NPY_INT64), NULL);
}

```

Рис. 3.11 - Згенерований код мовою програмування C функції count

Розглянемо час виконання розглянутої функції на стандартному Python інтерпретаторі та аналогічної функції, яка написана з допомогою примітивів NumPy та показана на рисунку 3.12.

```
1  def numpy_count(values, thresh):
2      return np.sum(values < thresh)
3
```

Рис. 3.12 - Реалізація функції count з допомогою примітивів NumPy

У таблиці 3.1 представлено найкращій середній час з 5 запусків на CPython, NumPy та за допомогою розробленого методу на 10 мільйонах випадково згенерованих наборах вхідних даних.

Таблиця 3.1 - Час виконання різних версій функції count

CPython	NumPy	Розроблений метод
31.5431s	0.0313s	0.0117s

У порівнянні з CPython, час виконання функції count з використанням розробленого методу динамічної компіляції у декілька тисяч разів менший. Не зважаючи на те, що розглянутий приклад досить простий, отриманий час виконання в декілька разів нижчий у порівнянні з простим використанням попередньо скомпільованих бібліотечних функцій NumPy. Головною причиною цього є те, що на виразах порівняння величин у NumPy виділяється масив для зберігання проміжних результатів, у той час як розроблений метод виконує операцію порівняння без таких виділень.

3.2 Реалізовані оператори та опис їх застосування

3.2.1 Прості вирази

Далі будуть розглянуті прості вирази, їх використання та особливості відтворення у проміжному представленні. Прості вирази не відносяться до створення, модифікації або перегляду значень масиву. До таких виразів відносяться створення чисел, кортежів, замикання та зрізи.

- *Const* (значення)

  Константами можуть бути дані булевого типу, цілі числа, числа з плаваючою комою або об'єкт None

- *Var* (ім'я)

  У проміжному представленні змінні мають походити з локального присвоєння у вигляді параметру, що передається до функції, SSA присвоєння або бути асоційовані з виразом контролю виконання коду.

- *PrimCall* (базова арифметична операція, аргументи)

  Базовими арифметичними операціями є такі оператори як додавання або ділення, також логічні операції як наприклад логічна операція або, також до базових арифметичних операцій відносяться математичні функції знаходження експоненту, синусу тощо.

- *Select* (умова, істинне значення, не істинне значення)

  Якщо умова яка є першим аргументом цього виразу є істинною або не істинною, то повертаються відповідні значення які також передаються до цього виразу.

- *Tuple* (елементи)

  Створюється кортеж (список, елементи якого не можна змінити) з n елементів, переданих як аргументи

- *TupleElt* (кортеж, індекс)

  Повертається значення кортежу відповідно до переданого індексу. Індекс має бути константою і не може бутии динамічно змінюючоюся величиною.

- *Closure* (функція, аргументи)

  Часткове передання n аргументів до функції, яка має мати хоча б n+1 вхідних параметрів. Результатом є об'єкт замикання, який може бути викликаний як функція.

- *ClosureElt* (замикання, індекс)

  Повертає з об'єкту замикання один з частково переданих аргументів до функції по вказаному індексу. Індекс має бути константою і не може бути динамічно змінюючоюся величиною.

- *Call* (функція, аргументи)

  Виклик функцій або об'єкту замикання з передачею аргументами відповідно до вказаних.

- *Slice* (початок, кінець, крок)

  Створюється зріз з масиву, списку або кортежу, початок і кінець це індекси елементів з яких починається і закінчується зріз, також можна вказати крок через який будуть відбиратися елементи.

3.2.2 Оператори присвоєння та контролю виконання програмного коду

Присвоєння встановлюють зв'язок між назвою змінної та її значенням. У плані контролю виконання не структуровані переходи по коду не підтримуються, тобто такі конструкції мови Python як break та continue не можуть використовуватися.

- *Assign* (об'єкт, значення)

  Обчислення значення та присвоєння його об'єкту, який може бути ім'ям змінної, значенням яке знаходиться по індексу масиву або кортежу.

- *If* (умова, блок правдивості умови, блок неправдивості умови)

  Вибіркове виконання коду блоків правдивості або неправдивості умови в залежності від булевого значення умови.

- *ForLoop* (комірка, початок, кінець, крок, тіло)

  Повторне виконання блоку коду, який знаходиться у тілі циклу, для ряду величин що зберігаються у відповідній комірці. Дані величини знаходяться у діапазоні ітерації від початку до кінцю з кроком. Даний вираз є еквівалентом конструкції for x in xrange (start,stop,step) мови Python.

- *WhileLoop* (умова, тіло)

  Повторне виконання блоку коду, який знаходиться у тілі циклу, до того моменту, поки умова стане не правдивою.

- *ParFor* (функція, межі)

  Виконання вказаної функції на елементах з кожним індексом, що знаходиться у межах значень від початкових до кінцевих. На відміну від звичайних циклів, цей оператор гарантує відсутність взаємодій між ітераціями, отже кожен виклик до функції може бути виконаний паралельно з іншими. Оператори над масивами вищого рівня, такі як Map та OuterMap понижуються до рівня оператору ParFor.

3.2.3 Оператори визначення параметрів масиву

- *Shape* (масив)

  Повертається кортеж з даними про розмір відповідного масиву.

- *Len* (масив або кортеж)

  Якщо до даного оператору передаєтьсямасив, то виконується виклик оператору Shape і відповідно повертається кортеж з даними про розмірність цього масиву, а у випадку коли передається кортеж повертається кількість елементів даного кортежу.

- *Rank* (об’єкт)

  Повертається число, яке вказує на розмірність переданого масиву, по іншому даний оператор можна описати як

Len(Shape(масив)), для всіх інших типів даних повертається число 0.

- *Strides*(масив)

  Повертається кортеж, який містить числовий зсув елементів по осям переданого масиву. Дані числа використовуються для обчислення адрес елементів масиву. Цей вираз відрізняється від аналогічного у NumPy тим, що він обчислюється за допомогою визначення номеру елементу, а не кількістю байт зсуву. Наприклад, адреса у пам'яті елементу масиву X [i, j, k] знаходиться так:базова адреса X + розмір елементу*(Strides(X) * i + Strides(X) * j + Strides(X) * k).

3.2.4 Базові оператори роботи з масивами

Такі оператори використовуються для створення нових масивів, перегляду існуючих масивів та простих перетворень. Багато з цієї операторів існують лише на ранній стадії компіляції і потім вони понижуються до конструкцій вигляду обчислених переглядів даних масиву, спеціалізованих використань операторів роботи з масивами вищого ривня та циклів.

- *AllocArray* (розмірність, тип елементів)

  Виділення пустого масиву з розмірність що передана у вигляді кортежу та також з елементами вказаного типу. Даний вираз є еквівалентним функції empty бібліотеки NumPy та використовується багатьма іншими операторами роботи з масивами після пониження їх рівня.

- *ConstArray* (значення, розмірність)

  Створення масиву заданої розмірності всі значення якого дорівнють переданому значенню. Використовується у реалізації NumPy функцій np.zeros та np.ones

- *Reshape* (масив, розмірність)

  Створення відображення існуючого масиву з його елементами, але іншої розмірності. Кількість елементів нового відображення має співпадати з кількістю елементів існуючого масиву.
- *Transpose* (масив)

  Створення відображення існуючого масиву з його елементами, проте зі зберіганням розмірності та кортежу зсувів елементів.
- *Ravel* (масив)

  Перетворення багатовимірного масиву у одновимірний вектор.
- *FromDiagonal* (вектор, розмірність, зсув)

  Конструюється масив більшої розмірності використовуючи переданий вектор, елементи якого становляться елементами діагоналі створеного масиву, значення всіх інших елементів дорівнює 0. По замовчанню вихідний масив є двохвимірною матрицею, проте за допомогою параметру розмірності цей параметр може бути зміненим. Якщо застосовується аргумент зсуву, тоді значення початкового вектору копіюються у діагональ, що є вищою на значення зсуву ніж основна діагональ. Якщо значення параметру є негативним, то значення вектору копіюється нижче основної діагоналі на відповідний зсув.
- *ExctractDiagonal* (масив, зсув)

  За допомогою даного виразу береться переданий масив та повертається вектор що містить діагональ цього масиву. Якщо застосовується аргумент зсуву, тоді у вектор копіюється діагональ, що є вищою на значення зсуву ніж основна діагональ. Якщо значення параметру є негативним, тоді у

вектор копіюється діагональ що нижче основної діагоналі на відповідний зсув.

- *Tile* (масив, повторення)

  Повторення вмісту переданого масиву по відповідній розмірності стільки разів, скільки вказано у аргументі повторення.

3.2.5 Оператори роботи з пам’яттю

Розглянуті далі конструкції є доступними для розробника, проте вони використовуються на більш пізніх етапах компіляції.

- *Alloc* (кількість елементів, тип елементів)

  Повертається вказівник на щойно створений буфер даних з кількістю елементамів та типом, що вказані у аргументах, розмір даного буферу даних обчислюється як множення розміру одного елементу певного типу у байтах на кількість таких елементів.

- *Free* (вказівник)

  Звільнення у ручному режимі буферу даних, що знаходиться за адресою вказівника, переданого у вигляді аргументу.

3.2.6 Оператори високого рівня

Відтворення паралелізму даних у розробленому методі відрізняється від традиційного в деяких речах.

Різноманітність операторів. Найбільшою різницею у порівнянні з традиційною реалізацією паралелізму даних є різноманітність операторів, які використовуються для проміжного представлення. Багато мов використовують основні функції Map та Reduce, на відміну від них у розробленому методі використовується багато інших операторів, таких як IndexMap, OuterMap, Filter тощо. Жоден з цих операторів не є простим, це означає що їх неможливо відтворити за допомогою комбінації базових операторів, проте це є і перевагою, так як не завжди можна реалізувати

оптимізації виконання за допомогою мінімалістичного підходу і за рахунок цього втрачаються такі можливості оптимізації.

Декілька масивів у якості аргументів одночасно. Оператори паралелізму даних зазвичай створені таким чином, що вони приймають єдину коллекцію, яка скомбінована за допомогою такого оператору високого рівня, як наприклад zip. На відміну від цього оператори роботи масиву у розробленому методі можуть напряму приймати декілька масивів у вигляді аргументів. Такі аргументи можуть навіть бути різної розмірності, наприклад за допомогою оператору вигляду Map(функція, матриця, вектор, вісь обходу = 0) буде застосовано передану функцію для кожного рядку матриці і всіх елементів вектора.

Вказання вісі обходу. Так як основним типом даних є n-вимірний масив існує необдність вказання вісі обходу для операторів паралелізму даних. Це реалізується за рахунок відповідного параметру у кожному операторі.

Додаткові параметри функцій. Зазвичай такі оператори, як Reduce та Scan спеціалізуються лише за допомогою бінарного оператора, такий підхід є неможливим у одночасному існуванні з підходом передачі декількох масивів у якості аргументів. Також необхідно вказати трансформацію, котра відбувається при застосуванні функції до всіх елементів (оператор map) вхідних масивів на акумулюючий тип.

- *Map* (функція, масив аргументів, вісь)

  Виконання функції для кожного елементу масиву аргументів. По замовчанню значення вісі дорівнює None, що означає що функція f виконується на всіх числових елементах масиву аргументів. Якщо у якості аргументу вісі буде вказано цілочисельне значення, тоді у якості аргументів до вказаної функції будуть передані зрізи відповідної розмірності. Це може бути використано для застосування функції до всіх рядків або стовпців матриці.

- *OuterMap* (функція, масив аргументів, вісь)

  Застосування функції для комбінацій елементів між вхідними масивами аргументів.
- *Filter* (функія, функція предикату, масив аргументів, вісь)

  Для вхідного масиву аргументів конструюється вихідний масив, що складається з результатів застосування переданої функції у випадку, коли функція предикату для відповідного елементу є правдивою, у іншому випадку значення відкидається.
- *IndexMap* (функція, кортеж розмірності)

  Даний оператор є схожим на оператор Map, у випадку IndexMap передана функція застосовується для значень індексів, що знаходяться у кортежі розмірності.
- *IndexFilter* (функція, функція предикату, кортеж розмірності)

  Даний оператор є схожим на IndexMap, проте у даному випдаку для відбору елементів, що знаходяться у кортежі розмірності застосовується функція предикату.
- *Reduce* (функція, масив аргументів, початкове значення, вісь)

  Комбінування усіх елементів масиву аргументів використовуючи рекурсивну передану функцію. Є можливість вказати початкове значення для переданої рекурсивної функції. Прикладами даного оператору є реалізація NumPy функцій sum та product, коли застосовується даний оператор для додавання або множення всіх елементів переданого масиву, запускається рекурсивна функція яка відповідно рекурсивно додаває або множить усі елементи (комбінує їх між собою).

- *IndexReduce* (функція, кортеж розмірності)

  Комбінування усіх елементів масиву аргументів використовуючи рекурсивну передану функцію. Комбінування виконується у індексних рамках відповідно до переданого кортежу розмірності.
- *FilterReduce* (функція, функція предикату, масив аргументів)

  Відбувається фільтрування елементів вхідних масивів використовуючи функцію предикату, далі виконується комбінування всіх залишившихся елементів за допомогою переданої рекурсивної функції.
- *Scan* (функція, масив аргументів, початкове значення, вісь)

  Відбувається комбінування всіх елементів масиву вхідних аргументів та повертається масив, у якому знаходяться усі проміжні дані обчислень від застосування рекурсивної функції. Є можливість вказати початкове значення для переданої рекурсивної функції. По замовчанню значення вісі дорівнює None, що означає що функція f виконується на всіх числових елементах масиву аргументів. Якщо у якості аргументу вісі буде вказано цілочисельне значення, тоді у якості аргументів до вказаної функції будуть передані зрізи відповідної розмірності. Це може бути використано для застосування функції до всіх рядків або стовпців матриці.
- *IndexScan* (функція, кортеж розмірності)

  Відбувається застосування оператору Scan для всіх індексів, що знаходяться у межах кортежу розмірності.

3.3 Висновок до третього розділу

У даному розділі проводиться огляд, принципи та особливості реалізації розробленого методу динамічної компіляції, що представляє собою надбудову над стандартним інтерпретатором мови Python та

перехоплює виконання позначених декоратором функцій. Функція проходить декілька етапів проміжних перетворень, також перед генерацією машиного коду відбувається декілька етапів оптимізації.

Організація роботи методу відбувається таким чином: на початку виконання програмного коду функція, що обернута декоратором, перетворюється у не типізоване проміжне представлення, далі коли вона викликається з певними аргументами відбувається типізація усіх даних відповідно до типів переданих у функцію аргументів, потім код проходить декілька етапів оптимізації і відбувається генерація C коду, який у подальшому передається на компілятор, що перетворює цей код у машинні інструкції відповідно до вибраного бекенду виконання програмного коду.

Також у цьому розділі розглянуто реалізовані оператори, при використанні яких генерується ефективний код, а також за допомогою яких реалізується паралелізм обробки даних, за рахунок якого досягається швидкодія виконання коду вища, ніж при використанні мов програмування, що застосовують звичайну компіляцію програмного коду без додаткових оптимізацій. Набір реалізованих операторів паралелізму даних є досить обширним, що дозволяє використовувати їх у багатьох випадках.

За рахунок типізації даних, використання операторів паралелізму даних та багатьох оптимізацій коду досягається значне пришвидшення у порівнянні зі стандартними засобами мови Python.

4 ТЕСТУВАННЯ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Застосування розробленого методу для різних алгоритмів

У даному розділі буде розглянуто застосування розробленого методу динамічної компіляції Python програм, орієнтованих на обробку масивів на наборі алгоритмів з інтенсивними обчисленнями. Нижче представлений вихідний програмний код кожного з алгоритмів та результати замірів його

швидкості виконання на стандартному інтерпретаторі мови Python - CPython, та результати виконання на трьох впроваджених бекендах: компілятору мови C, що використовує одне ядро та за допомогою технологій OpenMP і CUDA, що дозволяють сповна задіяти обчислювальні ресурси центрального процесора та графічного прискорювача відповідно.

Ступінь прискорення швидкості виконання програмного коду в основному залежить від того скільки часу витрачається на інтерпретацію цього коду. У гіршому випадку, коли більшу частину часу виконання програмного коду займає виконання низькорівневих високо-оптимізованих операторів, тоді результати можуть вийти повільнішими за отримані стандартними засобами мови Python. Проте з іншого боку, коли велика кількість елементів даних переглядається за допомогою чистого коду, застосування розробленого методу може пришвидшити швидкість виконання до декількох тисяч разів.

Також, при попередній оцінці швидкості прискорення виконання програмного коду за допомогою розробленого алгоритму необхідно враховувати фактор можливості застосування операторів паралелізму даних. Якщо алгоритм по більшій частині складається з операцій над масивами або з бібліотечних функцій NumPy, тоді застосування операторів паралелізму даних задіюється сповна, відповідно ефект від компіляції на бекендах OpenMP та CUDA є максимальним.

По замовчанню у якості бекенду використовується OpenMP. У деяких випадках можливо отримати значне прискорення швидкості виконання на графічному процесорі, за допомогою технології CUDA. Проте у інших випадках виконання програмного коду на графічному прискорювачі може бути менш ефективним у випадках великої кількості операції з пам'яттю, що є слабким місцем у графічних прискорювачах, та неефективною генерацією коду за допомогою технології CUDA. У деяких випадках алгоритми навіть не можуть бути запущенними на графічному

прискорювачі взагалі, якщо вони для свого виконання потребують більше системної пам'яті, ніж загалом доступно.

Першим алгоритмом у тестуванні є алгоритм Growcut. Цей алгоритм оснований на автоматній сегментації зображення, він приймає зображення та вказану користувачем маску та повторяє автоматне правило еволюції до отримання кінцевої сегментації. Програмний код даного алгоритму на мові Python наведений у рисунку 4.1

```
@jit
def growcut_python(image, state, window_radius):
    height = image.shape[0]
    width = image.shape[1]
    def attack(i,j):
        pixel = image[i, j, :]
        winning_colony = state[i, j, 0]
        defense_strength = state[i, j, 1]
        for jj in xrange(max(j-window_radius,0), min(j+
            window_radius+1, width)):
            for ii in xrange(max(i-window_radius, 0), min(i+
                window_radius+1, height)):
            if ii != i or jj != j:
                d = np.sum((pixel - image[ii, jj, :])**2)
                gval = 1.0 - np.sqrt(d) / np.sqrt(3)
                attack_strength = gval * state[ii, jj, 1]
                if attack_strength > defense_strength:
                    defense_strength = attack_strength
                    winning_colony = state[ii, jj, 0]
        return np.array([winning_colony, defense_strength])
    return np.array([[attack(i,j)
                      for i in xrange(height)]
                      for j in xrange(width)])

```

Рисунок 4.1 - Програмний код алгоритму Growcut

У таблиці 4.1 показано час виконання цього алгоритму для вхідного зображення розміром 600x600. Даний результат показує найкращий випадок прискорення відносно Python. Така низька продуктивність у Python пов'язана з відсутністю динамічної компіляції.

Таблиця 4.1 - Результати тестування швидкодії виконання алгоритму Growcut

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	1268.745	-	
C	0.478	2654	0.714
OpenMP	0.134	9468	0.889
CUDA	0.029	43749	2.428

Алгоритм перемноження матриць є одним із найбільш оптимізованих алгоритмів у всіх числових обчисленнях. Реалізація перемноження матриць у мові Python довжиною у одну строку коду вказана на рисунку 4.2, дана реалізація не включає в себе ніяких оптимізацій або засобів покращення продуктивності.

```
1  def matmult(X,Y):
2      return np.array([[np.dot(x,y) for y in Y.T] for x in X])
3
```

Рис. 4.2 - Програмний код перемноження двох матриць

Показані у таблиці 4.2 результати часу виконання отримані у результаті перемноження вхідних матриць розміру 1000x1000 з елементами типу float32. Результати часу виконання з використанням стандартного інтепретатору мови Python є досить швидкими, в основному через те, що перемноження чисел відбувається за допомогою ефективної реалізації за допомогою функції np.dot бібліотеки Numpy. Також для порівняння у таблиці вказаний результат суто Python реалізації перемноження чисел з використанням вбудованої у Python функції sum.

Таблиця 4.2 - Результати тестування швидкодії виконання перемноження матриць

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	17.40	-	-
C	12.19	1.4	0.34
OpenMP	3.85	4.5	0.29
CUDA	0.11	158.2	1.95

Наступна функція на якій відбулося тестування є градієнтним спуском функції Розенброка. Ця функція широко застосовується для визначення ефективності нелінійних методів оптимізації. Реалізація з використанням операцій над масивами NumPy вказана на рисунку 4.3. Так як арифметичні обчислення над масивами відбуваються через попередньо скомпільовані примітиви, застосування методу динамічної компіляції не надає такої сильної переваги як у випадку коли основні обчислення відбуваються всередині циклів, що оброблюються інтепретатором Python.

```
1  def rosenbrock_gradient(x):
2      der = np.empty_like(x)
3      der[1:-1] = (+ 200 * (x[1:-1] - x[:-2] ** 2)
4                   - 400 * (x[2:] - x[1:-1] ** 2) * x[1:-1]
5                   - 2 * (1 - x[1:-1]))
6      der[0] = -400 * x[0] * (x[1] - x[0] ** 2) - 2 * (1 - x[0])
7      der[-1] = 200 * (x[-1] - x[-2] ** 2)
8      return der
9
```

Рис. 4.3 - Програмний код знаходження градієнту функції Розенброка

У таблиці 4.3 вказані результати часу виконання цього коду на вхідному масиві, розміром у 10 мільйонів елементів типу float64.

Таблиця 4.3 - Результати тестування швидкодії знаходження градієнту функції Розенброка

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	0.4361	-	-
C	0.0631	6.9	0.275
OpenMP	0.0195	22.4	0.261
CUDA	0.008	54.5	2.249

Алгоритми згортки зазвичай обробляють матрицю невеликого розміру для підвищення різкості зображення, розмиття зображення, знаходження границь зображення тощо. Такі алгоритмі обчислюють нове значення вибраного пікселя, враховуючи значення пікселей, що його оточують. На рисунку 4.4 розглянутий програмний код застосування простого згорткового фільтру розміром 3x3 для кожного піксельного розташування у зображенні.

```
1  def conv_3x3(image, weights):
2      def pixel_result(i,j):
3          total = 0
4          for ii in xrange(3):
5              for jj in xrange(3):
6                  total += image[i-ii+1, j-jj+1] * weights[ii, jj]
7      return np.array([[pixel_result(i,j)
8                        for j in xrange(image.shape[1]-1)]
9                        for i in xrange(image.shape[0]-1)])
10
11
```

Рис. 4.4 - Програмний код згорткового фільтру розміром 3x3

Таблиця 4.4 - Результати тестування швидкодії застосування згорткового фільтру 3x3

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	10.972	-	-
C	0.017	645	0.761
OpenMP	0.008	1371	0.873
CUDA	0.003	3657	2.323

Звичайна лінійна регресія найменших квадратів із застосуванням єдиної незалежної змінної має досить просту реалізацію у результаті співвідношення наступних факторів: коваріації вхідних та вихідних даних та дисперсії вхідних даних. Реалізація за допомогою бібліотечних функцій NumPy призводить до того, що код дуже швидко інтерпретується, відповідно можна не очікувати значного прискорення виконання. (Рисунок 4.5).

```
1  def covariance(x,y):
2      return ((x-x.mean()) * (y-y.mean())).mean()
3
4  def fit_simple_regression(x,y):
5      slope = covariance(x,y) / covariance(x,x)
6      offset = y.mean() - slope * x.mean()
7      return slope, offset
8
```

Рис. 4.5 - Програмний код лінійної регресії

Проте навіть незважаючи на те, що інтерпретація відбувається дуже швидко, швидкість виконання програмного коду всеодно вийшло прискорити за рахунок паралельного виконання, результати наведено у таблиці 4.5.

Таблиця 4.5 - Тестування швидкодії виконання лінійної регресії

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	0.380	-	
C	0.056	6.8	0.353
OpenMP	0.014	27.1	0.401
CUDA	0.029	13.1	2.396

Алгоритм транспонування матриці являє собою отримання вихідної матриці з вхідної у результаті заміни рядків на стовпці. Вкладеність циклів показана на рисунку 4.6 реалізує транспонування матриці 4 рангу. Також більш компактний варіант реалізації з використанням операторів NumPy показаний на рисунку 4.7.

```
def rotate(T, g):
    def compute_elt(i,j,k,l):
        total = 0
        for ii in range(n):
            for jj in range(n):
                for kk in range(n):
                    for ll in range(n):
                        gij = g[ii, i] * g[jj, j]
                        gkl = g[kk, k] * g[ll, l]
                        total += gij * gkl * T[ii,jj,kk,ll]
        return total
    return np.array([[[[compute_elt(i,j,k,l)
                        for l in xrange(n)]
                        for k in xrange(n)]
                        for j in xrange(n)]
                        for i in xrange(n)])

```

Рис. 4.6 - Програмний код транспонування матриці

```
def rotate(T, g):
    gg = np.outer(g, g)
    gggg = np.outer(gg, gg).reshape(4 * g.shape)
    axes = ((0, 2, 4, 6), (0, 1, 2, 3))
    return np.tensordot(gggg, T, axes)

```

Рис 4.7 - Транспонування матриці використовуючи оператори NumPy

Результати транспонування матриці розміром 1000x1000x1000x1000 з елементами типу float64 наведені у таблиці 4.6

Таблиця 4.6 - Результати тестування швидкодії транспонування матриці

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	108.72	-	
C	0.057	1907	0.449
OpenMP	0.015	7248	0.578
CUDA	0.006	18120	2.581

Алгоритм Гаріса (рисунок 4.8) розпізнавання кутів та інших відмінних ознак на зображенні широко застосовується у багатьох інших алгоритмах обробки зображення, що є частиною комп'ютерного зору. В даному прикладі, більшість обчислень відбувається за допомогою попердньо скомпільованих примітивів бібліотеки NumPy, отже прискорення відбуваються в основному за рахунок поєднання операцій над масивами та паралельному виконанні, що дає не дуже великий приріст швидкодії, результати тестування подані у таблиці 4.7.

```
1  def harris(I):
2      m,n = I.shape
3      dx = (I[1:, :] - I[:m-1, :])[:, 1:]
4      dy = (I[:, 1:] - I[:, :n-1])[1:, :]
5      A = dx * dx
6      B = dy * dy
7      C = dx * dy
8      tr = A + B
9      det = A * B - C * C
10     k = 0.05
11     return det - k * tr * tr
12
```

Рис. 4.8 - Програмний код алгоритму Гаріса

Таблиця 4.7 - Тестування швидкодії виконання алгоритму Гаріса

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	0.173	-	
C	0.066	2.6	0.397
OpenMP	0.021	8.2	0.325
CUDA	0.004	43.2	2.23

Алгоритм гідродинаміки (рисунок 4.9.1 та рисунок 4.9.2) згладжених частинок був створений для симуляції поведінки рідин та газів. Алгоритм працює шляхом поділу рідини на дискретні елементи, які називають частинками. Ці частинки мають просторову відстань, на якій їх властивості «згладжуються» функцією ядра. Це означає, що кожна з фізичних величин кожної з частинок може бути отримана сумуванням відповідних величин усіх частинок, котрі перебувають у межах двох згладжених довжин.

```
1  def kernel_func(d, h) :
2      if d < 1: f = 1.-(3./2)*d**2 + (3./4.)*d**3
3      elif d<2: f = 0.25*(2.-d)**3
4      else: f = 0
5      return f/(np.pi*h**3)
6
7  def distance(x,y,z):
8      return np.sqrt(x**2+y**2+z**2)
9
10 def physical_to_pixel(xpos,xmin,dx):
11     return int32((xpos-xmin)/dx)
12
13 def pixel_to_physical(xpix,x_start,dx):
14     return dx*xpix+x_start
15
```

Рис. 4.9.1 - Перша частина алгоритму гідродинаміки згладжених частинок

```
15
16  def render_image(xs, ys, zs, hs, qts, mass, rhos, nx, ny, xmin, xmax, ymin,
17      ymax):
18      MAX_D_OVER_H = 2.0
19      image = np.zeros((nx,ny))
20      dx = (xmax-xmin)/nx
21      dy = (ymax-ymin)/ny
22      x_start = xmin+dx/2
23      y_start = ymin+dy/2
24      kernel_samples = np.arange(0, 2.01, .01)
25      kernel_vals = np.array([kernel_func(x,1.0) for x in kernel_samples])
26      qts2 = qts * mass / rhos
27      for i, (x,y,z,h,qt) in enumerate(zip(xs,ys,zs,hs,qts2)):
28          if ((x > xmin-2*h) and (x < xmax+2*h) and
29          (y > ymin-2*h) and (y < ymax+2*h) and
30          (np.abs(z) < 2*h)):
31          if (MAX_D_OVER_H*h/dx < 1 ) and (MAX_D_OVER_H*h/dy < 1):
32              xpos = physical_to_pixel(x,xmin,dx)
33              ypos = physical_to_pixel(y,ymin,dy)
34              xpixel = pixel_to_physical(xpos,x_start,dx)
35              ypixel = pixel_to_physical(ypos,y_start,dy)
36              dxpix, dypix, dzpix = [x-xpixel,y-ypixel,z]
37              d = distance(dxpix,dypix,dzpix)
38              if (xpos>0) and (xpos<nx) and (ypos>0) and (ypos<ny) and (d/h<2):
39                  kernel_val = kernel_vals[int(d/(.01*h))]/(h*h*h)
40                  image[xpos,ypos] += qt*kernel_val
41          else :
42              x_pix_start = int32(physical_to_pixel(x-MAX_D_OVER_H*h,xmin,dx))
43              x_pix_stop = int32(physical_to_pixel(x+MAX_D_OVER_H*h,xmin,dx))
44              y_pix_start = int32(physical_to_pixel(y-MAX_D_OVER_H*h,ymin,dy))
45              y_pix_stop = int32(physical_to_pixel(y+MAX_D_OVER_H*h,ymin,dy))
46              if(x_pix_start < 0): x_pix_start = 0
47              if(x_pix_stop > nx): x_pix_stop = int32(nx-1)
48              if(y_pix_start < 0): y_pix_start = 0
49              if(y_pix_stop > ny): y_pix_stop = int32(ny-1)
50              for xpix in range(x_pix_start, x_pix_stop):
51                  for ypix in range(y_pix_start, y_pix_stop):
52                      xpixel = pixel_to_physical(xpix,x_start,dx)
53                      ypixel = pixel_to_physical(ypix,y_start,dy)
54                      dxpix, dypix, dzpix = [x-xpixel,y-ypixel,z]
55                      d = distance(dxpix,dypix,dzpix)
56                      if (d/h < 2):
57                          kernel_val = kernel_vals[int(d/(.01*h))]/(h*h*h)
58                          image[xpix,ypix] += qt*kernel_val
59      return image
```

Рис. 4.9.2 - Друга частина алгоритму гідродинаміки згладжених частинок

Результати тестування швидкодії для рендеру зображень розміром 120x120 пікселів з 1600 частинками наведено у таблиці 4.8. Нажаль дана реалізація не могла бути протестована на бекендах з паралельним виконанням коду, отже часи виконання цього алгоритму для них відсутні.

Таблиця 4.8 - Результати тестування швидкодії виконання алгоритму згладжених частинок

	Час виконання (с)	Прискорення відносно Python	Час компіляції (с)
Python	11022.29	-	-
C	1.04	10,598	0.737

4.2 Висновок до четвертого розділу

У даному розділі розглянуто застосування розробленого методу динамічної компіляції Python програм, орієнтованих на обробку масивів на наборі алгоритмів з інтенсивними обчисленнями.

На конкретних прикладах алгоритмів, а саме: Growcut, перемноження матриць, знаходження градієнту функції Розенброка, застосування згорткового фільтру, виконання лінійної регресії, транспонування матриці, алгоритму Гаріса та алгоритму гідродинаміки згладжених частинок підтверджено, що ступінь прискорення швидкості виконання програмного коду в основному залежить від того скільки часу витрачається на інтерпретацію цього коду. Також експериментальним шляхом виявлено, що у загальному випадку час виконання алгоритмів на графічному процесорі менший, ніж на центральному процесорі, проте зачасту це пришвидшення нівелюється більш довгим етапом компіляції через неефективну генерацію машинного коду за допомогою технології CUDA.

ВИСНОВКИ

У даній роботі запропоновано спосіб динамічної компіляції масивно орієнтованих Python програм, що дозволяє розробнику писати код на мові Python, використовуючи популярну бібліотеку NumPy для математичних обчислень, орієнтованих на масиви, досягаючи продуктивності рівня високопродуктивних мов нижчого рівня на сучасному обладнанні. Розробнику не доводиться переходити на мови програмування нижчого рівня, що призводить до втрат часу та появи можливих помилок. Завдяки використанню розробленого методу, мова Python стає потужним інструментом для вирішення задач аналізу даних та наукових обчислень, так як вдається позбавитися головного недоліку інтерпретації у байт-код — низької у порівнянні з мовами нижчого рівня, такими як C або Fortran, швидкості виконання обчислювально інтенсивних алгоритмів.

Для забезпечення високої продуктивності відбувається ряд оптимізацій коду: усунення спільних виразів, згортка констант, видалення мертвого коду, запроваджена реалізація операторів паралелізму даних та відбувається типізація функцій для кожного набору вхідних аргументів, що у сукупності дає значне поліпшення швидкодії у порівнянні зі стандартними засобами мови Python. Також відбувається процес вбудування функцій, що дозволяє позбавитися недоліку реалізації операторів роботи над масивами бібліотеки розширення масивів NumPy, а саме виділення тимчасових об'єктів для зберігання проміжних результатів, так як оператори NumPy є попередньо скомпільованими. Також деякі оператори NumPy повторно реалізовані з використанням паралелізму даних, що дозволяє задіяти усю обчислювальну здатність сучасного багатоядерного апаратного забезпечення, центральних процеросів та графічних прискорювачів. Впроваджена підтримка бекендів, за допомогою яких розробнику надається можливість вибирати апаратуру, на якій буде виконуватися машинний код у паралельних потоках.

У даній роботі детально розглянуто реалізацію запропонованого методу і всі етапи перетворення початкового програмного коду на шляху до генерації машинних інструкцій. Організація роботи методу відбувається таким чином: на початку виконання програмного коду функція, що обернута декоратором, перетворюється у не типізоване проміжне представлення, далі коли вона викликається з певними аргументами відбувається типізація усіх даних відповідно до типів переданих у функцію аргументів, потім код проходить декілька етапів оптимізації і відбувається генерація коду мови програмування C, який у подальшому передається на компілятор, що перетворює цей код у машинні інструкції відповідно до вибраного бекенду виконання програмного коду.

Створений метод динамічної компіляції протестовано на наборі складних у плані обчислень алгоритмів, де він показав на реальних результатах значне поліпшення у швидкодії виконання програмного коду. Максимальне прискорення швидкості виконання вдалося досягнути у тих алгоритмах, виконання яких в основному відбувається інтерпретатором мови Python. Проте навіть якщо основна частина обчислень виконується на попередньо скомпільованих примітивах бібліотеки NumPy, за рахунок їх реалізації з використанням паралелізму обробки даних, уникання марних виділень проміжних значень за рахунок вбудування операторів, що виконують операції над масивами, також вдалося досягти вагамого прискорення виконання коду.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

Сравнение интерпретатора, обычного и JIT компиляторов [Електронний ресурс]. — Режим доступу : https://tproger.ru/translations/interpreter-compiler-jit/

1. Разница между компилятором и интерпретатором [Електронний ресурс]. — Режим доступу : http://devnuances.com/soft/raznitsa-mezhdu-kompilyatorom-i-interpretatorom/
2. Огляд процесу компіляції програмного коду[Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%96%D0%BB%D1%8F%D1%82%D0%BE%D1%80
3. Огляд процесу інтерпретації програмного коду[Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80
4. Лутц, М. Изучаем Python. 4-е издание [Текст] / Марк Лутц : ISBN: 978-5-93286-159-2. — 1272 с.
5. Документація бібліотеки розширення масивів NumPy [Електронний ресурс]. — Режим доступу : https://docs.scipy.org/doc/numpy-1.14.2/reference/
6. Конференція присвячена науковим обчисленням мовою Python [Електронний ресурс]. — Режим доступу : https://scipy2017.scipy.org/ehome/index.php?eventid=220975&
7. Порівняння та огляд машинного коду та байт-коду [Електронний ресурс]. — Режим доступу : https://javarush.ru/groups/posts/394-mashinnihy-kod-i-bayt-kod-na-kakom-jazihke-govorit-vasha-programma
8. Байт-код [Електронний ресурс]. — Режим доступу : https://uk.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4

9. Введение в машинные коды [Електронний ресурс]. — Режим доступу : http://citforum.ck.ua/programming/windows/machine_code/

10. Компилированные и интерпретируемые языки [Електронний ресурс]. — Режим доступу : https://sozdaj-sam.com/javascript/kompilirovannye-i-interpretiruemye-jazyki.html

11. Как работает Python? [Електронний ресурс]. — Режим доступу : https://adw0rd.com/2009/08/22/python-howto-work/

12. Виртуальная машина Python (PVM) [Електронний ресурс]. — Режим доступу : http://python-3.ru/page/virtualnaja-mashina-python-pvm

13. AOT-компіляція [Електронний ресурс]. — Режим доступу : https://ru.wikipedia.org/wiki/AOT-%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F

14. Ahead-of-time компиляция видеолекция [Електронний ресурс]. — Режим доступу : https://www.lektorium.tv/lecture/27486

15. Введение в JIT компиляцию [Електронний ресурс]. — Режим доступу : http://www.polovko.me/blog/2012/10/04/vvedenie-v-jit-kompilyaciyu/

16. Динамическая компиляция и измерение производительности [Електронний ресурс]. — Режим доступу : https://www.ibm.com/developerworks/ru/library/j-jtp12214/index.html

17. Документація бібліотеки Psyco [Електронний ресурс]. — Режим доступу : http://psyco.sourceforge.net/introduction.html

18. Параллельное программирование на OpenMP [Електронний ресурс]. — Режим доступу : http://ccfit.nsu.ru/arom/data/openmp.pdf

19. Документація технології OpenMP [Електронний ресурс]. — Режим доступу : http://www.openmp.org/resources/tutorials-articles/

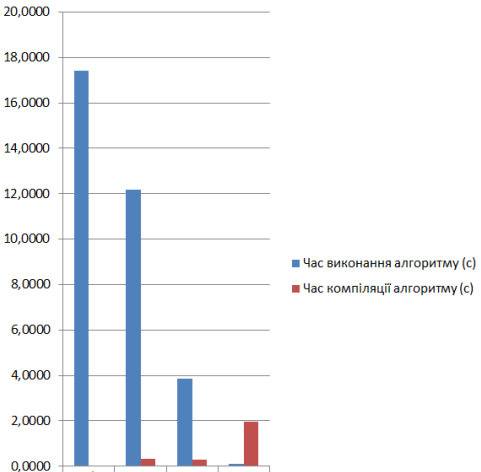
20. Документація технології CUDA [Електронний ресурс]. — Режим доступу : http://www.nvidia.com.ua/object/cuda-parallel-computing-ru.html

Додатки

Додаток 1
Копії графічних матеріалів

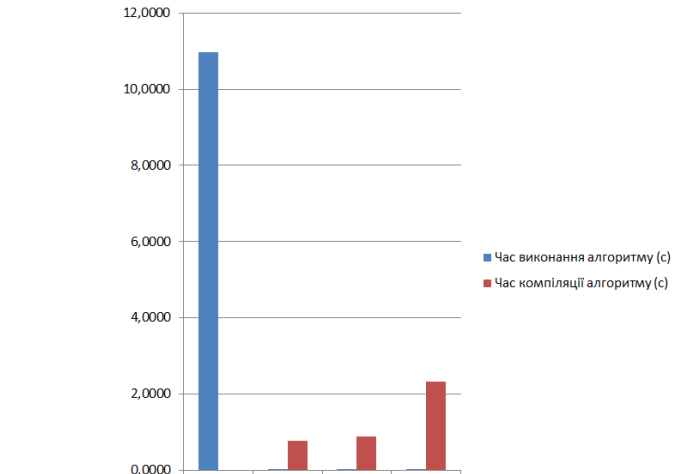
Демонстраційні гістограми результатів тестування методу на наборі алгоритмів

Результати тестування швидкодії виконання перемноження матриць розміру 1000x1000 з елементами типу float32



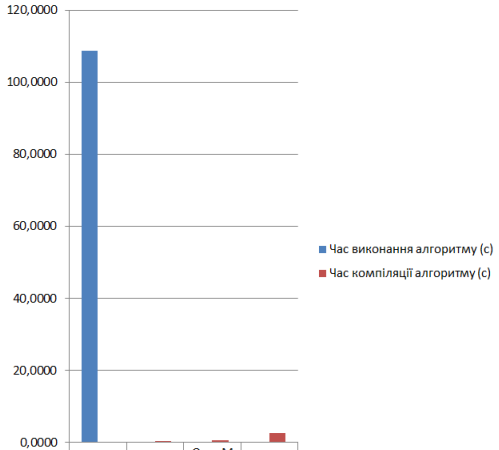
	Python	C	OpenMP	CUDA
Час виконання алгоритму (с)	17,4000	12,1900	3,8500	0,1100
Час компіляції алгоритму (с)	0,0000	0,3400	0,2900	1,9500

Результати тестування швидкодії застосування згорткового фільтру розмірністю 3x3



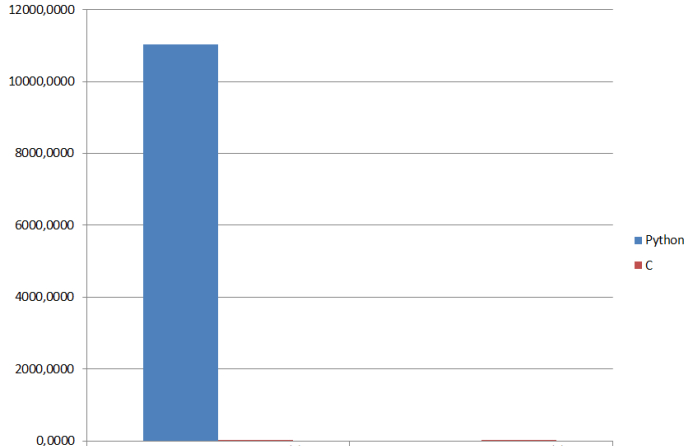
	Python	C	OpenMP	CUDA
Час виконання алгоритму (с)	10,9720	0,0170	0,0080	0,0030
Час компіляції алгоритму (с)	0,0000	0,7610	0,8730	2,3230

Результати тестування швидкодії транспонування матриці розміром 1000x1000x1000x1000 з елементами типу float64



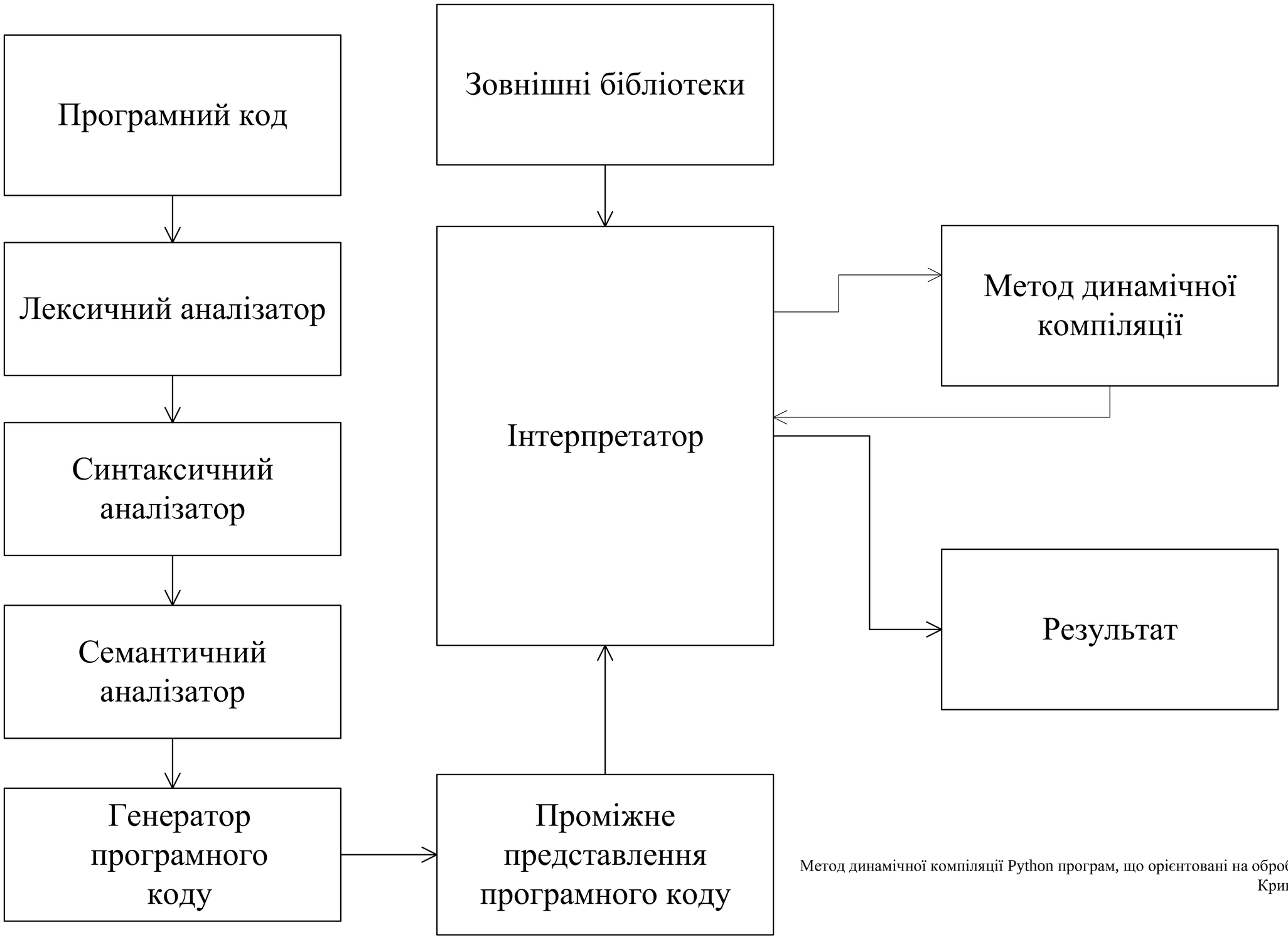
	Python	C	OpenMP	CUDA
Час виконання алгоритму (с)	108,7200	0,0570	0,0150	0,0060
Час компіляції алгоритму (с)	0,0000	0,4490	0,5780	2,5810

Результати тестування швидкодії виконання алгоритму згладжених частинок для рендеру зображення розміром 120x120 пікселів з 1600 частинками



	Час виконання алгоритму (с)	Час компіляції алгоритму (с)
Python	11022,2900	0,0000
C	1,0400	0,7370

Структурна схема процесу інтерпретації програмного коду з використанням запропонованого методу



Метод динамічної компіляції Python програм, що орієнтовані на обробку масивів
Кривомаз М. Є.

Схема алгоритму оптимізації програмного коду функції перед утворенням проміжного нетипізованого представлення

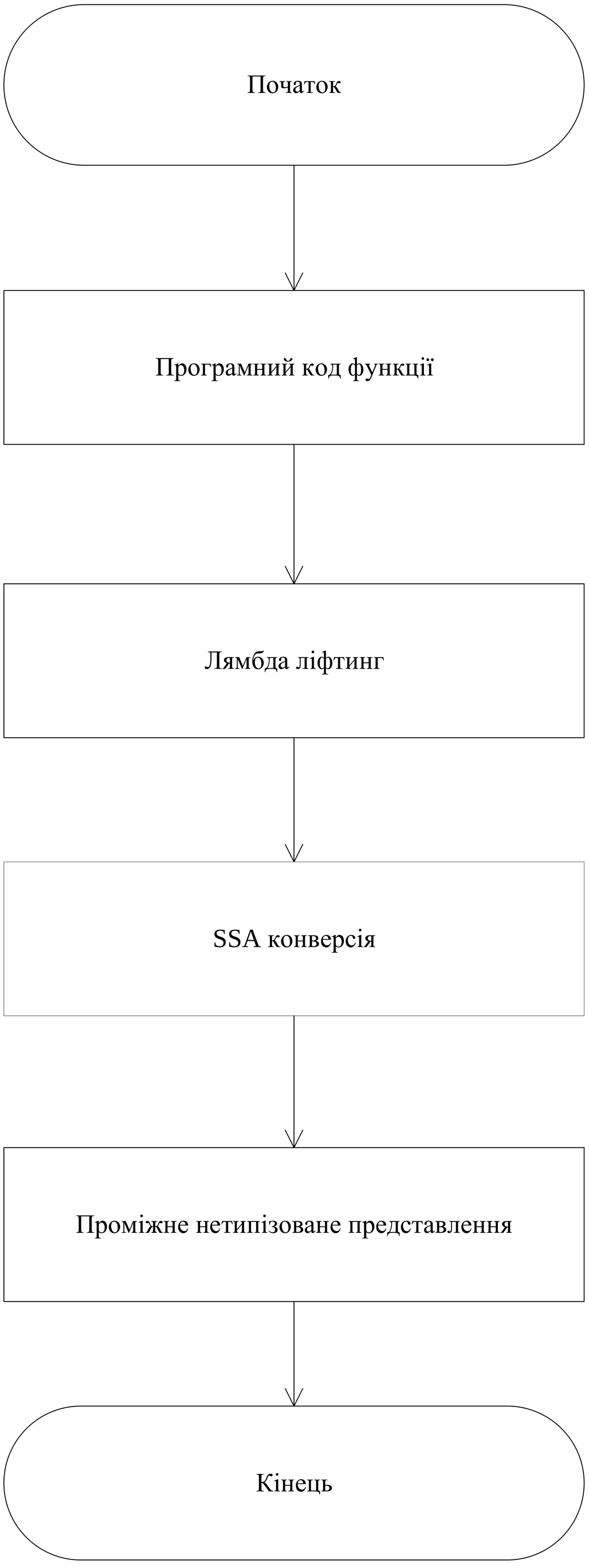


Схема алгоритму оптимізації нетипізованого представлення після виклику функції

Схема алгоритму роботи запропонованого методу

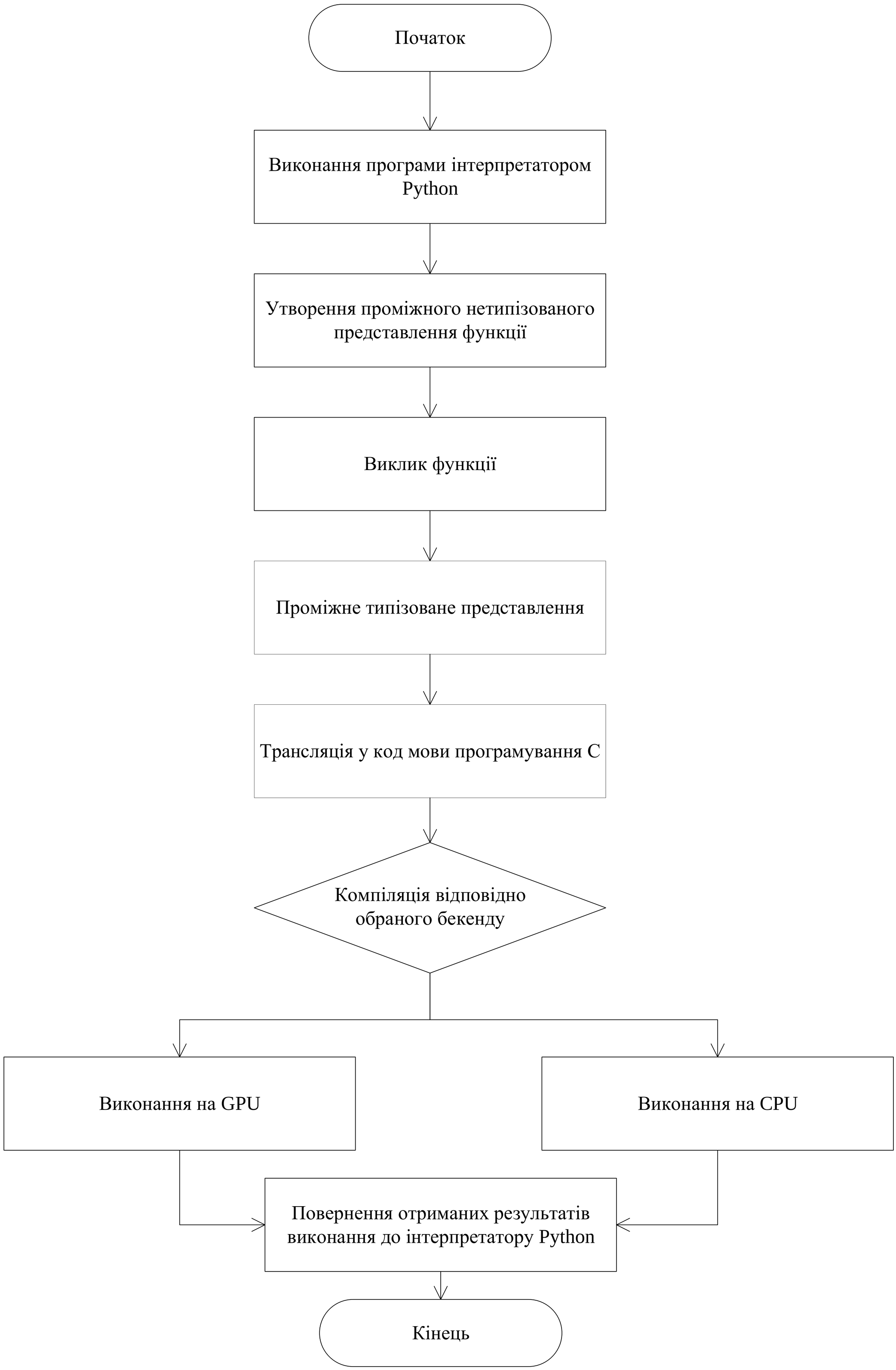
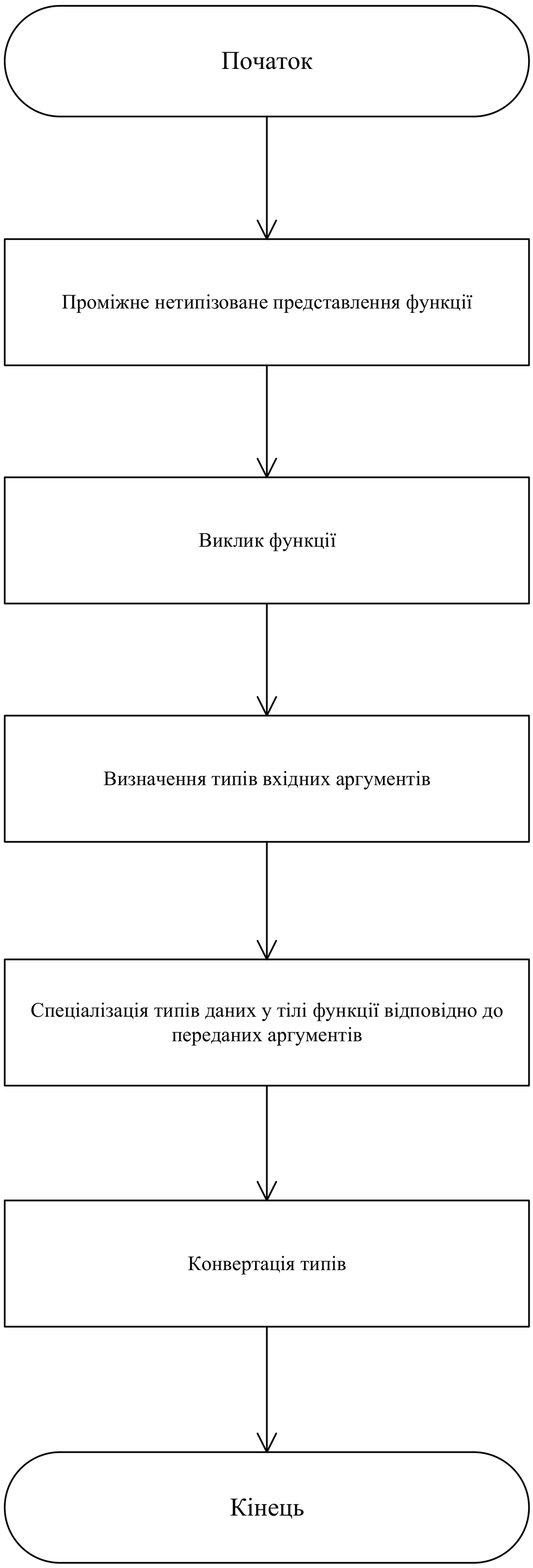


Схема алгоритму типізації даних у функції



Метод динамічної компіляції Python програм, що орієнтовані на обробку масивів
Кривомаз М. Є.

Додаток 2
Копії публікацій по темі магістерської дисертації

УДК 004.441

Д.т.н., доцент Терейковський І.А., студент Кривомаз М.Є.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

СПОСІБ ДИНАМІЧНОЇ КОМПІЛЯЦІЇ МАСИВНО ОРІЄНТОВАНИХ PYTHON ПРОГРАМ

Abstract

Igor A. Terejkovskij, assoc. prof., PhD; Maxim Krivomaz, student
Method of dynamic translation of array oriented Python programs

This paper concerns the method of dynamic translation of array oriented Python programs. This method bridges the gap between the usability of Python and the speed of code written in efficiency languages. Most attention in this paper is paid to performance issues and ways to solve them.

Вступ

Мова програмування Python стала популярною платформою для аналізу даних і наукових обчислень. Для вирішення проблеми низької продуктивності стандартного інтерпретатора мови Python, математично інтенсивні обчислення, як правило, переносяться на бібліотечні функції, які написані на високопродуктивних мовах програмування, таких як Fortran або C. Якщо така бібліотека для виконання певного алгоритму відсутня, програмісту доводиться прийняти низьку продуктивність або перейти на мову нижчого рівня для ефективної реалізації поставленої задачі.

Таким чином, у процес розробки проекту входить етап прототипування алгоритмів на мови програмування нижчого рівня, а потім безпосередньо відбувається процес переносу розділів з низькою продуктивністю у такій мові як Python, на мову нижчого рівня. Цей етап може займати багато часу, призводити до появи помилок і відводить увагу розробника від початкової задачі.

Постановка задачі

Створити спосіб динамічної компіляції масивно орієнтованих Python програм для вирішення проблеми низької продуктивності математично інтенсивних обчислень у мові Python та уникнення прототипування

алгоритмів на мови нижчого рівня та переносу розділів з низькою продуктивністю на такі мови. Розроблений спосіб надає можливість поєднати зручність використання мови Python та забезпечить швидкість виконання коду як на ефективних мовах програмування. Провести експерименти порівняння часу виконання ряду алгоритмів на різних апаратних засобах.

Термінологія

Динамічна компіляція – технологія збільшення продуктивності програмних систем, що використовують байт-код, шляхом компіляції байт-коду в машинний код або в інший формат безпосередньо під час роботи програми.

Інтерпретатор – програмний засіб, що виконує аналіз, обробку та виконання вихідного коду програми або запиту.

Опис способу

Розроблений спосіб є бібліотекою для прискорення математичних операцій у мові Python, написаних з використанням розширення масивів NumPy [1]. Спосіб доповнює середу виконання Python, не заміняючи її. Для запуску функції з використанням можливостей бібліотеки необхідно позначити її декоратором @PAR. Для прикладу розглянемо наступний код, написаний з використанням NumPy для знаходження середнього значення з трьох масивів:

```
@PAR
def avg(x,y,z):
     return (x+y+z) / 3.0
```

При вказанні декоратору @PAR відбувається перехват викликів до відміченої функції та створюється типізована версія даної функції. Процес типізації відбувається, коли до функції передається аргументи певного типу, тоді відбувається клонування тіла функції, визначаються типи для кожної змінної у відповідності до переданих до функції аргументів та відбувається приведення типів, у випадку якщо їх необхідно конвертувати. Динамічна компіляція досягається за рахунок того, що функція, позначена декоратором, підготовується для подальшої компіляції, та у момент, коли вона викликається відбувається процес типізації для переданих до неї параметрів, а вже потім вона виконується. Вказання типів збільшує швидкість виконання коду, так як, наприклад, додати дві змінні типу float, що знаходяться у відповідних регістрах у рази швидше ніж це виконується

звичайними засобами мови Python, а саме виклик операції __add__ для двох об’єктів PyFloatObjects.

Якщо усунути декоратор, то функція avg виконається як звичайний Python код. Так як функції з бібліотеки NumPy компілюються окремо, вони завжди виділяють масив, який є результатом, навіть коли він одразу ж використовується. На відміну від цього, розроблений спосіб допомагає уникнути непотрібних виділень, виконує типізацію функцій для кожного типу вхідних даних, та виконує код у вигляді паралельних потоків.

Паралельні потоки виконання коду досягаються за рахунок використання наступних операторів [2]:

- map *(f, X_1, ..., X_n, pattern=None)* – виконання функції *f* для кожного елементу масиву аргументів. *Pattern* може використовуватися для того, щоб задати певний патерн ітерацій, наприклад для того, щоб виконати функцію *f* для всіх рядків або стовпців;

- allpairs *(f, X_1, X_2, pattern=0)* – виконання функції *f* для кожної пари елементів з масивів X_1 та X_2;

- reduce *(f, X_1, ..., X_n, pattern=None)* – комбінування всіх елементів масиву аргументів використовуючи рекурсивну функцію *f*;

- scan *(f, X_1, ..., X_n, pattern=None)* – комбінування всіх елементів масиву аргументів та повернення результату у вигляді масиву, що містить усі накопичені проміжні значення обчислень.

Для вище описаних операторів синтезується код, що виконується у паралельних потоках, та безпосередньо виконується вказаний оператор у відповідності з його аргументами. Розглянемо їх використання на прикладі нескладної функції:

```
@PAR
def add1 (x):
     return x + 1
```

Якщо функція add1 була викликана з цілочисельним аргументом, тоді вона буде скомпільована для повернення цілочисельного результату. Але надалі у випадку коли у неї буде передано аргумент у вигляді числа з плаваючою комою, тоді буде створена нова її реалізація для повернення результату з плаваючою комою. При передачі у дану функцію аргумента у вигляді вектора, буде згенеровано спеціалізовану версію функції з використанням оператора map:

```
def add1_map (x):
     return map (lambda xi : xi +1 , x)
```

Також код проходить декілька етапів оптимізації [3], які допомагають уникнути надлишкових обчислень, а саме:

- видалення спільних виразів – відбувається пошук обчислень, які виконуються більше одного разу на ділянці коду, що розглядається, такі обчислення видаляються та їх результати замінюються збереженними результатами першого обчислення;

- вбудування функцій – відбувається пошук функцій, які можна поєднати між собою і виконати за допомогою лише одного виклику;

- згортка констант – відбувається обчислення константних виразів на етапі компіляції, перш за все спрощуються констатні вирази, які містять числові літерали.

Висновки

Розроблений спосіб динамічної компіляції масивно орієнтованих Python програм дозволяє розробнику писати код на мові Python, використовуючи популярну бібліотеку NumPy для математичних обчислень орієнтованих на масиви, досягаючи продуктивності рівня високопродуктивних мов нижчого рівня на сучасному обладнанні. Розробнику не доводиться переходити на мови програмування нижчого рівня, що призводить до втрат часу та появи можливих помилок.

Для забезпечення високої продуктивності відбувається ряд оптимізацій коду, використання операторів для паралельного виконання потоків та відбувається типізація функцій для кожного набору вхідних аргументів, що у сукупності дає значне поліпшення швидкодії у порівнянні зі стандартними засобами мови Python.

При порівнянні запропонованого способу зі звичайним середовищем виконання мови Python отримуємо значне підвищення швидкості виконання коду. Для прикладу розглянемо швидкість виконання перемноження двох матриць розмірністю 1200x1200 з типом елементів float32. З використанням стандартних засобів мови Python це відбувається за 16.6 секунд, а за допомогою розробленого способу це можливо здійснити всього за 4.5 секунди.

Література

1. Документація роботи з бібліотекою NumPy [Електронний ресурс]. NumPy Documentation Tutorial. Режим доступу: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html
2. N. Lincoln. Parallel programming techniques forcompilers. SIGPLAN Not., 5(10) pages 18–31, Oct. 2010.
3. C. Chen, J. Chame, and M. Hall. In Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO), pages 111–122, 2015.

УДК 004.441

Д.т.н., доцент Терейковський І.А., студент Кривомаз М.Є.
Національний авіаційний університет,
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»
Електронна пошта: on_trigger@ukr.net

ДИНАМІЧНА КОМПІЛЯЦІЯ PYTHON-ПРОГРАМ В ЗАДАЧАХ ЗАХИСТУ ІНФОРМАЦІЇ

В задачах захисту інформації мова програмування Python являється однією із найбільш популярних платформ, що використовуються для створення засобів інтелектуального аналізу даних. Особливістю такого аналізу є необхідність обробки великих масивів даних. При цьому проявляється характерний недолік мови Python, що полягає у низькій продуктивності його інтерпретатора. Для вирішення проблеми низької продуктивності стандартного інтерпретатора мови Python, математично інтенсивні обчислення, як правило, переносяться на бібліотечні функції, які написані на високопродуктивних мовах програмування, таких як Fortran або C. Якщо така бібліотека для виконання певного алгоритму відсутня, програмісту доводиться прийняти низьку продуктивність або перейти на мову нижчого рівня для ефективної реалізації поставленої задачі.

Таким чином, у процес розробки проекту входить етап прототипування алгоритмів на мови програмування нижчого рівня, а потім безпосередньо відбувається процес переносу розділів з низькою продуктивністю у такій мові як Python, на мову нижчого рівня. Цей етап може займати багато часу, призводити до появи помилок і відводить увагу розробника від початкової задачі.

Для виправлення вказаного недоліку розроблено модуль динамічної компіляції Python-програм. По суті цей модуль є бібліотекою для прискорення математичних операцій у мові Python, написаних з використанням розширення масивів NumPy. Ця бібліотека широко використовується для програмної реалізації алгоритмів захисту інформації, досягаючи

продуктивності рівня високопродуктивних мов нижчого рівня на сучасному обладнанні.

Модуль доповнює середу виконання Python, не заміняючи її. Для запуску функції з використанням можливостей бібліотеки необхідно позначити її декоратором @PAR. Для прикладу розглянемо наступний код, написаний з використанням NumPy для знаходження середнього значення з трьох масивів:

```
@PAR
def avg(x,y,z):
       return (x+y+z) / 3.0
```

За рахунок декоратора @PAR відбувається перехоплення викликів до відміченої функції та створюється типізована версія даної функції. Процес типізації відбувається, коли до функції передаються аргументи певного типу, тоді відбувається клонування тіла функції, визначаються типи для кожної змінної у відповідності до переданих до функції аргументів та відбувається приведення типів, у випадку якщо їх необхідно конвертувати.

Динамічна компіляція досягається за рахунок того, що функція, позначена декоратором, підготовлюється для подальшої компіляції, та у момент, коли вона викликається відбувається процес типізації для переданих до неї параметрів, а вже потім вона виконується. Вказання типів збільшує швидкість виконання коду, так як, наприклад, додати дві змінні типу float, що знаходяться у відповідних регістрах у рази швидше ніж це виконується звичайними засобами мови Python, а саме виклик операції __add__ для двох об'єктів PyFloatObjects.

Паралельні потоки виконання коду досягаються за рахунок використання наступних операторів [1]:

- map *(f,* X_1*, ...,* X_n*, pattern=None)* – виконання функції *f* для кожного елементу масиву аргументів. *Pattern* може використовуватися для того, щоб задати певний патерн ітерацій, наприклад для того, щоб виконати функцію *f* для всіх рядків або стовпців;

- allpairs *(f,* X_1*,* X_2*, pattern=0)* – виконання функції *f* для кожної пари елементів з масивів X_1 та X_2;
- reduce *(f,* X_1*, ...,* X_n*, pattern=None)* – комбінування всіх елементів масиву аргументів використовуючи рекурсивну функцію *f*;
- scan *(f,* X_1*, ...,* X_n*, pattern=None)* – комбінування всіх елементів масиву аргументів та повернення результату у вигляді масиву, що містить усі накопичені проміжні значення обчислень.

Для вище описаних операторів синтезується код, що виконується у паралельних потоках, та безпосередньо виконується вказаний оператор у відповідності з його аргументами. Розглянемо їх використання на прикладі нескладної функції:

```
@PAR
def add1 (x):
	return x + 1
```

Якщо функція add1 була викликана з цілочисельним аргументом, тоді вона буде скомпільована для повернення цілочисельного результату. Але надалі у випадку коли у неї буде передано аргумент у вигляді числа з плаваючою комою, тоді буде створена нова її реалізація для повернення результату з плаваючою комою. При передачі у дану функцію аргумента у вигляді вектора, буде згенеровано спеціалізовану версію функції з використанням оператора map:

```
def add1_map (x):
	return map (lambda xi : xi +1 , x)
```

Також код проходить декілька етапів оптимізації [2], які допомагають уникнути надлишкових обчислень, а саме:

– видалення спільних виразів – відбувається пошук обчислень, які виконуються більше одного разу на ділянці коду, що розглядається, такі обчислення видаляються та їх результати замінюються збереженними результатами першого обчислення;

– вбудування функцій – відбувається пошук функцій, які можна поєднати між собою і виконати за допомогою лише одного виклику

– згортка констант – відбувається обчислення константних виразів на етапі компіляції, перш за все спрощуються константні вирази, які містять числові літерали.

Висновки

Розроблений модуль динамічної компіляції Python- програм дозволяє розробнику писати код на мові Python, використовуючи популярну бібліотеку NumPy для математичних обчислень орієнтованих на масиви.

Для забезпечення високої продуктивності відбувається ряд оптимізацій коду, використання операторів для паралельного виконання потоків та відбувається типізація функцій для кожного набору вхідних аргументів, що у сукупності дає значне поліпшення швидкодії у порівнянні зі стандартними засобами мови Python.

Література

1. N. Lincoln. Parallel programming techniques forcompilers. SIGPLAN Not., 5(10) pages 18–31, Oct. 2010.

2. C. Chen, J. Chame, and M. Hall. In Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO), pages 111–122, 2015.